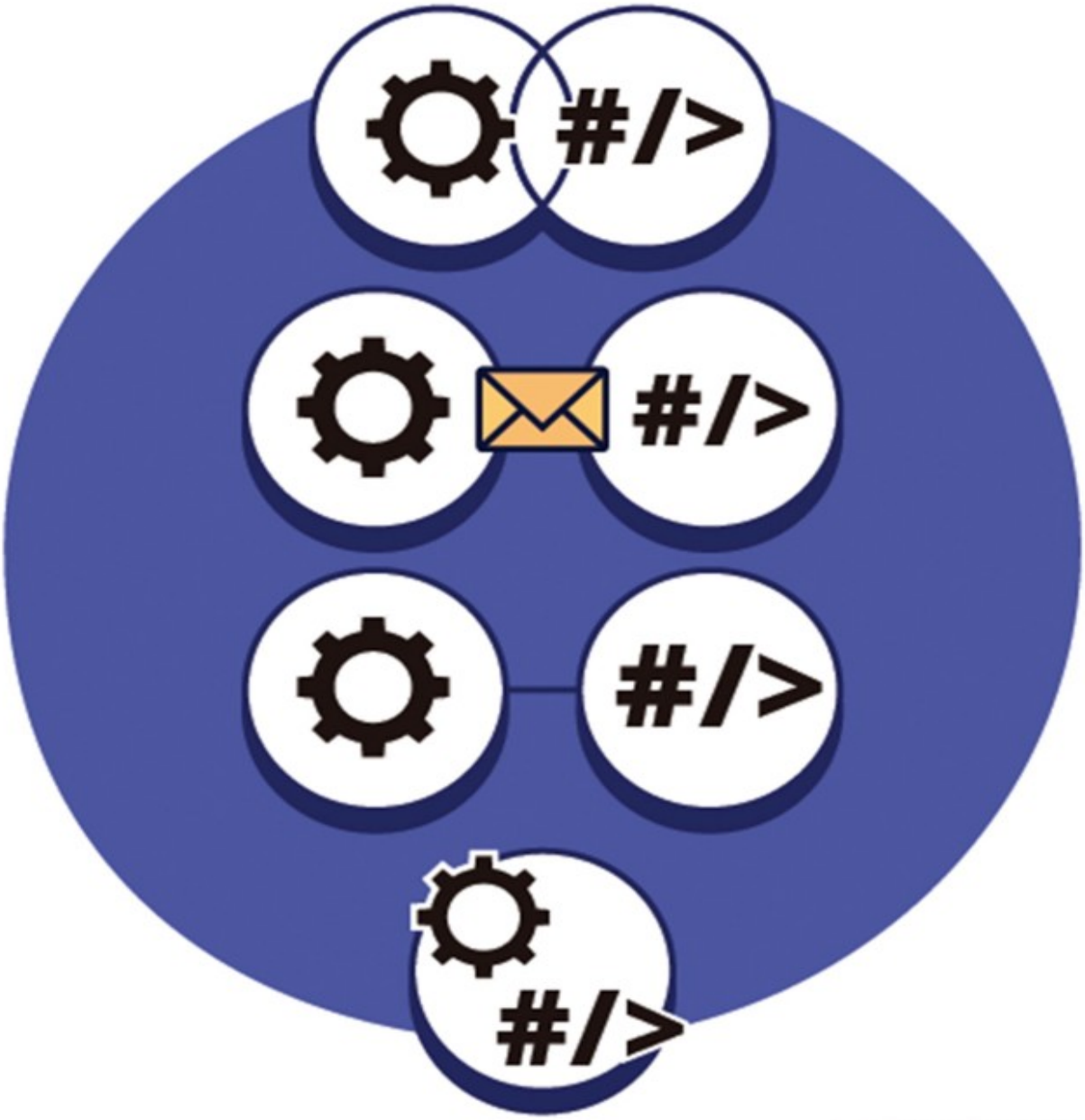
How DevOps works

Organizing people, from silos to platform teams



LEONARDO LEITE
PAULO MEIRELLES
FABIO KON

Casa do Código | alura

Sumário

- ISBN
- Preface
- Part 1: DevOps
  - 1 What is DevOps?
  - 2 Impacts of DevOps
  - 3 DevOps Challenges
  - 4 DevOps Tools
- Part 2: The different ways to organize DevOps
  - 5 Overview
  - 6 Segregated departments
  - 7 Collaborating departments
  - 8 Single departments
  - 9 API-mediated departments
  - 10 Enabler teams
  - 11 Organizational structures beyond DevOps
  - 12 Conclusions
  - 13 Appendix — Our scientific approach
  - 14 Bibliographic references

ISBN

Digital: 978-85-5519-392-7

Preface

When I started in the software industry 25 years ago, we didn’t have the cloud or DevOps. Back then, things were simpler, and maintaining an application or website was the responsibility of the *webmaster*. The webmaster is a super-specific example of the archetype defined in this book as a “single development and infrastructure department” — in this case, a one-person department. From designing and styling pages to programming business logic, to storing data in a database, maintaining and updating web services, email, and databases, the webmaster had to know how to do it all and keep it all running on the same “metal”.

In the first websites I created, we didn’t have redundancy and high availability. By the time we had a script to back up the database, and that was enough. Our customers weren't as demanding either. Pages with animated gifs and colorful "under maintenance" text flashing with the `<blink>` tag were common. Users would turn off their dial-up connections or go to another site and come back the next day, and that was all normal.

Today, with the cloud, most professionals in the software industry don't even know exactly where this "metal" is. The stack has evolved a lot, and so have the tools at our disposal. User expectations and demands have also increased. We want websites and applications that are always online, constantly and transparently updated, and without causing any disruption to our flows.

We can no longer expect a webmaster to be able to take care of everything, and that's why we saw the DevOps movement emerge, to make the lives of professionals in the software industry better, promoting collaboration between people who specialize in development and those who specialize in infrastructure.

But how do you do DevOps? This book can help software industry professionals gain a better understanding of DevOps and answer this question in a number of ways. Here you will find a clear and concise

definition of DevOps, as well as an overview of its key concepts and practices, and this can be useful both for professionals who are new to DevOps and for those who want to update their knowledge.

You will also understand the different ways to organize DevOps. The book identifies four different organizational models for DevOps: siloed departments, collaborative departments, single departments, and API-mediated departments. These definitions were not invented, they emerged from thorough research with professionals from a variety of organizations, and I am sure that if you are already on your journey, you will recognize some of these patterns. If you are just starting out, understanding the patterns will help you to choose the right approach for your organization, depending on your specific needs and constraints.

The book also highlights the challenges of DevOps, such as the need for cultural change and the need to overcome silos, and the potential benefits, such as increased speed and agility, improved quality, and reduced costs. This can help professionals to be aware of potential pitfalls and develop strategies to address them. The book includes several practical tips on how to implement DevOps, how to choose the right tools, and how to measure the success of DevOps initiatives, which can be useful for professionals who are ready to start implementing DevOps in their own organizations.

Some specific examples of how this book can help professionals in the software industry:

- Software engineers who are new to DevOps can use the book to learn about key DevOps concepts and practices. This can help them to understand how DevOps can be used to improve the software development process.
- Managers who are considering implementing DevOps in their organizations can use the book to learn about the different ways to organize for DevOps and the potential benefits and challenges of each approach. This can help them to choose the

right approach for their organization and develop an rollout plan.
- Team leaders who are already implementing DevOps in their teams can use the book to learn about new tools and techniques that can be used to improve their DevOps practices. The book also includes several case studies from real-world organizations that can provide inspiration and guidance.

I have worked in large companies with separate development and operations departments and suffered from seeing my code stuck in change management approval processes that did not add value. Here you can learn about the risks of the segregated department pattern and how to avoid them without having to suffer so much.

I have also worked in small startups where I and a few developers had skills in infrastructure and the responsibility to execute everything we build. This was great for increasing my knowledge of the entire stack, and it might be a good pattern to use if your context is similar. However, I also saw how this archetype did not allow startups to scale.

At Nubank, I led the platform team that first fell into the trap of recreating a silo and becoming overwhelmed, and then realized that the API relationship archetype was the best way to scale, leveraging our infrastructure talent to the fullest.

At Google, I see how the collaborative model between the development and infrastructure departments (what we call SRE) is a great model for scaling and that, in our case, also leverages API mediation.

In short, this book is an excellent resource for software industry professionals who want to learn more about DevOps and how to implement it in their own organizations. With rigorous research behind it, this is a highly recommended read for anyone interested in DevOps. I loved recognizing all the lessons I have learned

throughout my career, building and maintaining large distributed systems, consolidated here to accelerate your journey.

Alexandre Freire Kawakami

Director of Engineering for Privacy and Security at Google and head of the São Paulo office

Acknowledgements

This book is the result of research based on interviews with industry professionals. Therefore, we would first like to deeply thank the 75 incredible professionals with whom we had the opportunity to interact throughout this research. It is people like these who forge the spirit of collaboration in the DevOps movement.

We would also like to thank the researchers who, to our great pride, collaborated directly with our research: Dejan Milojicic (HP Labs Palo Alto), Carla Rocha (University of Brasília – UnB), Gustavo Pinto (Federal University of Pará – UFPA), Nelson Lago (University of São Paulo – USP) and Claudia Melo (University of São Paulo – USP).

Some colleagues willingly helped us revising the text of this book. Therefore, we would like to thank Renato Cordeiro, Antonio Terceiro, Arthur Pilone Maia da Silva, Lorenzo Bertin Salvador and Lucas Santana Santos! We are also grateful to the colleagues who worked directly on the production of this book: Paulo Henrique Albuquerque dos Anjos de Souza, Bruna Keese and our editor at Casa do Código, Vivian Matsui.

We would also like to thank the following funding agencies for their support: National Council for Scientific and Technological Development - CNPq (Process 465446/2014); Foundation for the Improvement of Higher Education Personnel - CAPES (Financial Code 001); and São Paulo Research Foundation - FAPESP (Processes 14/50937-1, 15/24485-9, and 19/12743-4). In addition, we would like to thank the Federal Data Processing Service (Serpro), employer

of Leonardo Leite, who led the aforementioned research and writing of this book.

Finally, we would like to thank our families for supporting our dedication to our careers, research, and writing of this book, especially our wives, Larissa Suyama, Larissa Negreiros, and Fernanda Estevan.

About the authors

Leonardo Leite

Leonardo Leite received his master's and doctorate degrees in computer science from the University of São Paulo (USP) in 2014 and 2022, respectively. Since 2014, he has been a software developer at the Federal Data Processing Service (Serpro), where he successfully supported the implementation of DevOps practices, such as the adoption of automated tests, deployment pipelines, continuous delivery, and monitoring.

In his master's research, he studied the automated deployment of service compositions. Deployment automation is one of the main technical concerns of DevOps. Leonardo also contributed to the free software project *Radar Parlamentar*, in which he applied DevOps techniques, especially continuous deployment. He has already published textbooks with Senac on databases.

Leonardo is the author of a heavy metal album inspired by the Divine Comedy (*Heaven's Sewer* channel on YouTube) and enjoys taking his 2-years-old son, Dante, for walks in the sling (baby wear). Leonardo values teleworking as it enables him to be closer to his son.

- Lattes: http://lattes.cnpq.br/5529201020890212
- LinkedIn: https://www.linkedin.com/in/leonardo-alexandre-ferreira-leite-phd-17194322
- Google Scholar: https://scholar.google.com.br/citations?user=Y8ZLD_kAAAAJ

Prof. Paulo Meirelles

Paulo Meirelles obtained his doctorate in computer science from the University of São Paulo (USP) in 2013. His teaching career has been unconventional, with positions at the University of Brasília (UnB), the Federal University of São Paulo (UNIFESP), and the Federal University ABC (UFABC). Since 2023, he has been a professor in the Department of Computer Science at IME-USP. Throughout his career, he has led software development projects, successfully implementing DevOps practices. Additionally, he has maintained a strong relationship and a history of collaboration with the free software community, focusing most of his software engineering research on free software.

Paulo loves sports and has a black belt in judo. He is the father of Fernando and Bento and dedicates his participation in this book to their mother, Larissa, his life partner.

- Lattes: http://lattes.cnpq.br/2193972715230641
- LinkedIn: https://www.linkedin.com/in/paulormm/
- Google Scholar: https://scholar.google.com.br/citations?user=LnO5njYAAAAJ

Prof. Fabio Kon

Fabio Kon graduated in music from the Paulista State University (UNESP) and in computer science from the Institute of Mathematics and Statistics at USP (IME-USP). He later earned his PhD in computer science from the University of Illinois at Urbana-Champaign. He is currently a full professor at IME-USP, where he teaches, supervises, and conducts research in software engineering, agile methods, distributed systems, data science, and smart cities. He is the coordinator of the National S&T Institute on the Future Internet for Smart Cities (https://interscity.org). He is the drummer of the band *Líricos Oníricos* and is seeking a jazz quartet to play his vibraphone. Is anyone qualified?

- Lattes: http://lattes.cnpq.br/2342739419247924
- LinkedIn: https://www.linkedin.com/in/fabiokon/

- Google Scholar: https://scholar.google.com.br/citations?user=VE8BW84AAAAJ

What is this book about?

DevOps is a very popular topic among IT professionals, with many books published and conferences on the subject. Even without a unanimous definition in the community as to what DevOps is, it is certain that the DevOps movement has indelibly impacted the software industry. Some of its practices, such as continuous delivery and the deployment pipeline (also called deployment mat), have already been consolidated as fundamental good practices for accelerating software delivery.

Through the study of academic articles and other sources, we, the authors of this book, have produced a synthesis of our understanding of DevOps, encompassing its history, concepts, and implications. This synthesis was published in the form of a scientific article in a high-impact international journal under the title *A Survey of DevOps Concepts and Challenges* (LEITE et al., 2019), having received a large number of citations (an indicator of success in the academic world). The first part of this book translates our synthesized conceptualization of DevOps into Portuguese. In particular, this summary addresses our definition of DevOps which we hope will be useful to the community.

In our "*Survey of DevOps*", we realized the topic raises a world of questions, some technical and others more human (or social). And among human issues, we noticed a crucial problem, relating to the organization between development and infrastructure teams, which was often not clear in the dominant literature until then. This problem was the target of doctoral research by Leonardo Leite, author of this book, conducted at the University of São Paulo (USP) and guided by professors Paulo Meirelles and Fabio Kon (also authors of this book). This was software engineering research conducted based on interviews with 75 professionals from 59 different companies from different countries. The second part of this book brings into Portuguese, in a more accessible way, the important

results of this research, already published in renowned international scientific journals. Furthermore, the book is filled with quotes taken from our interviews, which gives the book a special flavor and makes the reader feel present in the companies analyzed.

SOFTWARE ENGINEERING RESEARCH

Software engineering, as scientific research, is a field of study that analyzes not only the structure of software systems, but also how we, humans, social beings that we are, build these systems.

The problem: how to organize people in DevOps?

The DevOps movement assumes the existence of the roles of the developer person (*dev*) and the operator person (*ops*). In fact, the term "operator" is controversial, but let's leave that question for later in this book. What happens is that the automation promoted by the DevOps movement has blurred the definition of these roles, as well as the expectations in the relationships between them. Put another way, after adopting an automated deployment pipeline (one of the main DevOps practices, exemplified in another chapter later), what does *ops* do? Who sets up this pipeline? Who executes it? And how do these *ops* come to relate to the *devs*? The lack of clear and consensual answers ends up causing stressful discussions and negotiations in software-producing companies.

Different companies found different answers to the presented dilemma. At Amazon, for example, the team that develops a service is responsible for its operation (GRAY, 2006). It's what they call "*you build it, you run it*"). Google, on the other hand, is betting on highly qualified engineers to operate the services at an early stage. These people, called SREs (*site reliability engineers*), invest efforts to

reduce the operational overhead of the services, so that, later, the developers themselves can operate the services with minimal effort (BEYER, 2016).

The point here is that these two approaches are very different and have very different consequences, but both, depending on the context, can be labeled as DevOps. It seemed to us, therefore, that there was a lack of a more refined vocabulary to properly label these experiences and better debate their differences. Furthermore, there was no guarantee that there would not be other forms of organization in other companies.

Therefore, we assumed that there would be different ways of *organizing* the development and infrastructure teams. With "organization" (academically called "organizational structure"), we refer to the division of labor between teams and the expected interactions between them (OLIVEIRA, 2012), as illustrated in the following figure. In our research, we focused on analyzing the division of labor of operational activities and interactions in relation to development and infrastructure teams. For brevity, in this book, we will talk about “ways of doing DevOps” or even “DevOps structures”.

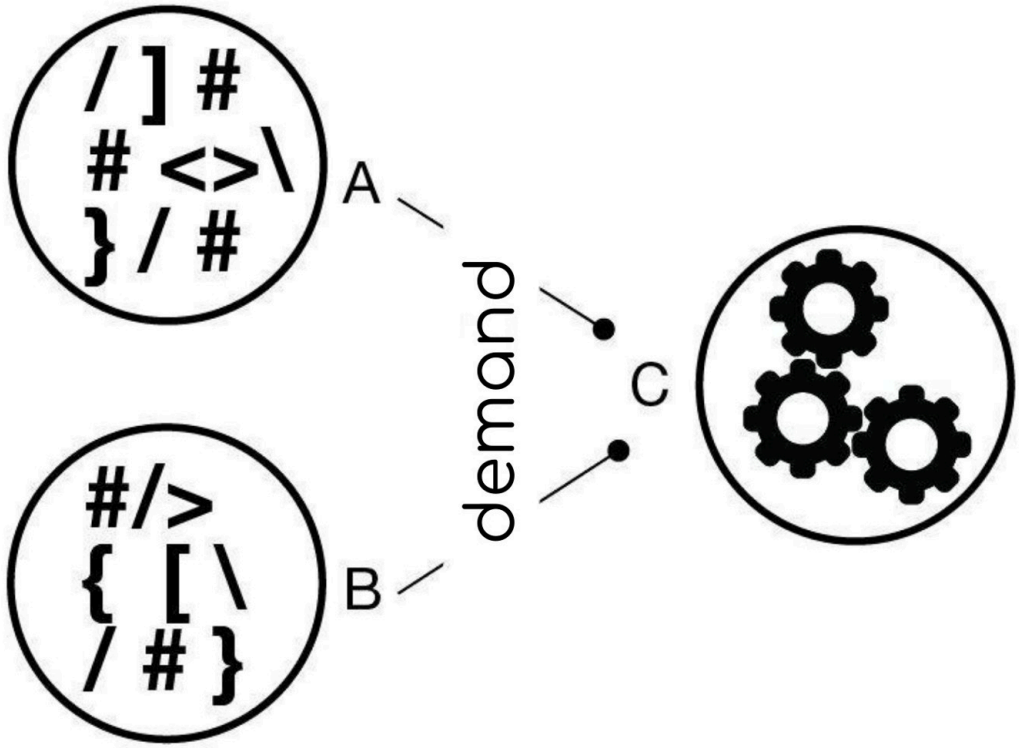


Figura 0.4: Example of an organization in which teams A and B develop and team C take care of the infrastructure; teams A and B demand services from team C and, ideally, teams A and B do not need to interact with each other

In short, the conclusion obtained in our research on ways to do DevOps consists of the following classification:

- **Segregated departments**, with highly bureaucratized interaction between development and infrastructure groups;
- **Collaborating departments**, focusing on communication and collaboration between development and infrastructure groups;
- **Single departments**, in which a cross-functional team (sometimes also called a product team) takes responsibility for software development and infrastructure management;
- **API-mediated departments**, where the infrastructure team (the platform team) provides highly automated infrastructure services to assist developers.

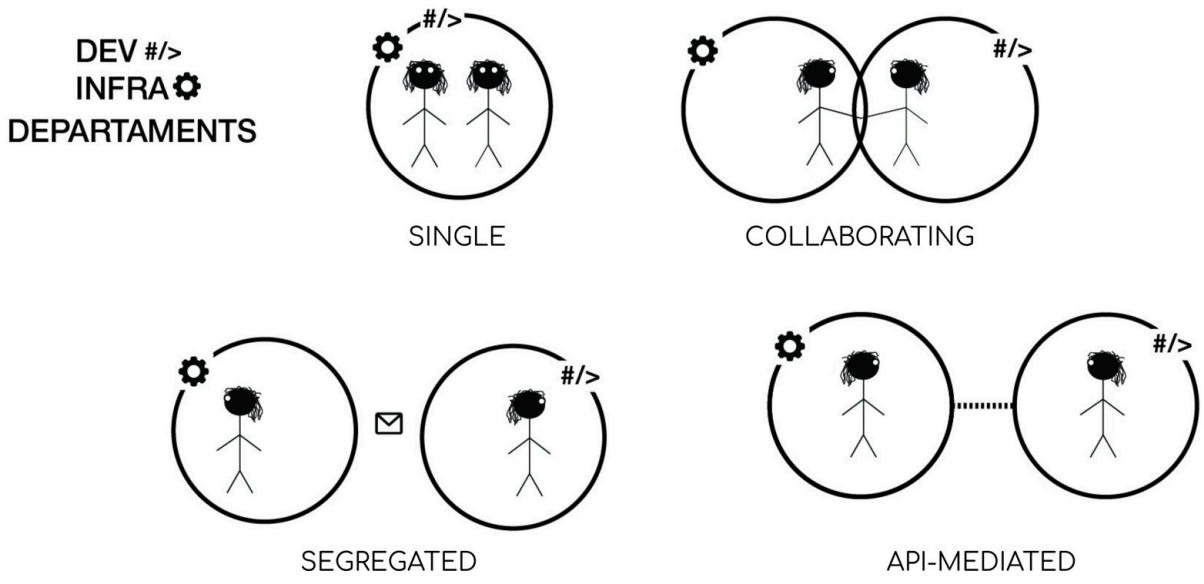


Figura 0.5: Pictographic representation of the four ways of doing DevOps identified in our research

It is worth mentioning that we did not start from the idea that there would be more correct or more mature ways of doing DevOps. However, it is very important that you understand that these different forms apply to different contexts and that they have different consequences (advantages and disadvantages). A quick example: a large company that wants to adopt a model similar to Amazon's will have the great challenge of ensuring that all development teams have people who master infrastructure management. In other words, while Amazon may praise the benefits of its model, if you want to adopt it in your business, you need to be aware of its inherent challenges and consider whether it is even suitable for your context.

This book is here to help you gain an *understanding* of the different ways of doing DevOps. This understanding comprises the conditions, causes, reasons to avoid, consequences, and how to deal with the disadvantages of each alternative. That's it: more than a recipe for "what to do", our book is here to offer you an understanding of "how" DevOps is done in different ways, in different companies; that is, we offer an explanation of the world. This derives, to a certain extent, from the academic origin of this work. Most popular books in

our field are written by consultants, so they have guidance language ("do this and that to achieve success"). In our research, we employ a scientific method to understand a certain phenomenon, in a similar way to what is done in sociology. While practical advice is also given in the book, the bottom line is that it is up to you to evaluate your context to determine how best to apply the ideas presented here.

Our classification of ways to do DevOps has the important benefit of clearly differentiating between the options of *collaborating departments* and *single departments*, both traditionally covered under the term DevOps. Furthermore, an important virtue of our classification is the presentation of a relatively new way of doing DevOps, in which developers interact with the so-called *platform team*. In particular, this way of working seems to enable a more frequent pace of software deliveries, which is one of the main goals of DevOps. This relationship is, in fact, an important result of our research. Furthermore, the literature on these platform teams is relatively scarce: before our research, there was practically no academic work on the topic, which is also practically unexplored in the Portuguese language (which motivated us to first publish this book in Portuguese, the authors' native language).

The different ways of doing DevOps present different solutions for the balance between specialization and interaction between teams. However, this balance is not an issue that only concerns DevOps, but the entire industry in general. And it's always good to know when your problems don't just concern your company or even just your industry. Therefore, to provide you with a broader perspective on the problem, we will also have a more than special chapter contextualizing this problem beyond the scope of the software. This chapter will provide a brief history of the issue, including Adam Smith, the construction of the Boeing 777, *Toyotism*, complex systems, and the discussion about culture, among other points.

Who is this book for?

This book proves to be particularly useful for the management team of companies' software producers. Its content can support discussions and decisions about a company's DevOps journey, including questions about team design: which teams should exist, how they should be composed, what their responsibilities should be, how they should interact with each other, and the challenges inherent to each possible solution.

As we believe that decisions such as those mentioned should not exclude the participation of workers who produce the software, especially senior-level professionals or those who wish to become so, this book can help you get involved in broader and more strategic decisions within your organization. We remember that *lean* principles, the basis of agile methods, preach that product quality and continuous improvement of processes is everyone's responsibility (POPPENDIECK, M.; POPPENDIECK, T., 2006).

Reading this book does not impose any strong technical requirements. We hope that readers who have already minimally participated in a software project will be fully able to follow the text. For beginners in the world of software production, some difficulties may arise due to a lack of knowledge of some basic terms in the area (for example, "production environment" and "deployment"). However, we have sought to clarify these terms throughout the book, so that this work is widely accessible to all people involved in software production.

Finally, we can say that this book is not for those who are just looking for better ways to "do their part", but for those who are looking to understand the collective effort of software production, as

well as promote improvements in this process in their context. We hope you like it!

Part 1: DevOps

In this first part of the book, we provide an introduction to DevOps, including its history, concepts, implications, challenges, and tools, as well as our definition of DevOps. The content presented here originates mainly from the article *A Survey of DevOps Concepts and Challenges* (LEITE et al., 2019), written by the authors of this book and published in *ACM Computing Surveys*, a scientific journal of high international impact. The chapters in this part also aim to prepare readers not yet accustomed to the topic of DevOps for the second part of the book, in which we present our original contribution to ways of doing DevOps.

Capítulo 1
What is DevOps?

Since the beginning of the DevOps movement in 2010, its proponents have advocated the use of culture, automation, measurement, and sharing as solutions to some of the software industry's problems. These problems would mainly be how to align teams in a company, how to deliver software more frequently, and how to have a stable production environment (HUMBLE; MOLESKY, 2011).

The lack of consensus on the definition of DevOps is one of the factors that has promoted a multiplicity of views within the DevOps movement. Thus, part of the community focused on adopting new automation tools, while others focused on the notion that DevOps would be a culture. This certainly causes some difficulty for beginners trying to understand what DevOps is. Because, if we stop to think, some terms widely used in this context are difficult to understand — for example, what is *culture*? As if that weren't enough, the software market even created the figure of the "DevOps engineer", which, for some people, strongly conflicts with the notion that DevOps would be some type of culture.

In this chapter, we will present our vision of what DevOps is. This vision includes our definition, which is anchored in academic studies on the topic and also on the history of DevOps, which is where we begin this chapter. Let's also clarify what is the "*ops*" in the DevOps, that is, the role of "operator".

Completing our vision, we will also present a set of concepts that surround DevOps and are organized into four groups: *process*, *people*, *delivery* and *runtime*. With these concepts in mind, we will be better equipped to discuss the different ways of doing DevOps.

1.1 History of DevOps

We consider DevOps to be an evolution of *agile software development*. Agile development advocates the existence of small cycles throughout a project so that there is the release of a new version of the system in each of these cycles; this way customers can review each intermediate version and guide development through their feedback. This scheme assumes that the team can frequently deploy the system in an production-like environment.

However, the pioneering literature on the agile movement ended up neglecting specific deployment practices. As evidence, we have that no Extreme Programming (XP) practice, for example, is about deployment.

This same lack of attention to deployment was also evident in so-called traditional software development processes, such as RUP (the Rational Unified Process). The RUP transition phase focused much more on procedures such as creating manuals and user training than on the technical activities of moving the system to the production environment.

RUP Phases

Let's quickly recall the software development phases foreseen in RUP:

- Initiation / Conception;
- Preparation (project/architecture);
- Construction (implementation);
- Transition (deployment).

We remember that these phases were not exactly sequential, which would configure the waterfall model, but they would have overlaps between them (one begins while the previous one is still ending).

AGILE MOVEMENT

The agile development movement broke with previously existing standards in software development. While previously traditional processes focused on rigorous planning, both from a management point of view and from a software architecture point of view, the Agile Manifesto (https://agilemanifesto.org/iso/ptbr/manifesto.html) proposes that we should value:

- *Individuals and interactions over processes and tools*;
- *Working software over comprehensive documentation*;
- *Customer collaboration over contract negotiation*;
- *Responding to change over following a plan*.

Kent Beck, signatory of the Agile Manifesto, proposed his own versoin of agile methods, XP (Extreme Programming). XP advocates a set of values, principles, and practices that differ from other agile approaches by explaining the importance of certain technical practices. Some of these practices involve the use of automated tests and continuous integration (the work of different developers must be unified daily), which are the basis for the deployment pipeline, the foundation of DevOps. The XP canon is in the book *Extreme Programming (XP) Explained*.

As a consequence of Software Engineering's negligence at the time in the deployment process, the transition to production tended to be a stressful process in organizations, containing manual and error-prone activities, and even last-minute corrections. If you've never experienced this experience, you can get an idea of this scenario by reading the first chapters of the book *The Phoenix Project*, an excellent book about DevOps, organized in the form of a narrative authored by Gene Kim (2018), one of the most prominents figures in the DevOps movement. Given all this context, we can understand

that DevOps came, in a complementary way to agile methods, to effectively enable the iterative delivery of software in short cycles.

From an organizational perspective, the DevOps movement promotes closer collaboration between developers and operators, avoiding the so-called **silos**, in which these two areas become well segregated. Traditionally, operations personnel are responsible for managing modifications to software at production and service levels; Development teams, on the other hand, are in charge of continually developing new features to meet business requirements. Thus, in a pre-DevOps situation, each of these departments has its processes, tools, and knowledge bases. In this situation, the interface between them is usually a *tickets* system: development teams demand the release of new versions of the software, while operations staff manually address these tickets.

In this arrangement, development teams want to release new versions to production frequently, while operations personnel try to prevent such changes, aiming for software stability and compliance with non-functional requirements. Supposedly, this structure would enable more stability for the software in production. In practice, however, it results in long delays between an update to the source code and its respective deployment. It also results in inefficient problem-solving processes, so that silos are more concerned with blaming each other than with preventing problems from occurring again.

These conflicts between developers and operators, plus significant delays in implementation, led to a scenario in which the promises of agile development were not fully fulfilled, as frequent deliveries, in each development cycle, were difficult to achieve. In this context, developers and operators began to collaborate within companies to solve this problem. This movement was coined DevOps in 2008, when Patrick Debois (2008) called for "agile infrastructure", considering the use of testing, frequent deployments, and the inclusion of infrastructure people in the product team. Soon after, John Allspaw and Paul Hammond (2009) highlighted how devs and

ops collaborated at Flickr (a popular photo-sharing site created in 2004) to achieve a high rate of deployments ("more than 10 per day") through automation of infrastructure, sharing of the versioning system, small deliveries, feature toggles, sharing of monitoring and other techniques that are precursors to the deployment pipeline, in addition to a culture that involves respect, trust and fault tolerance.

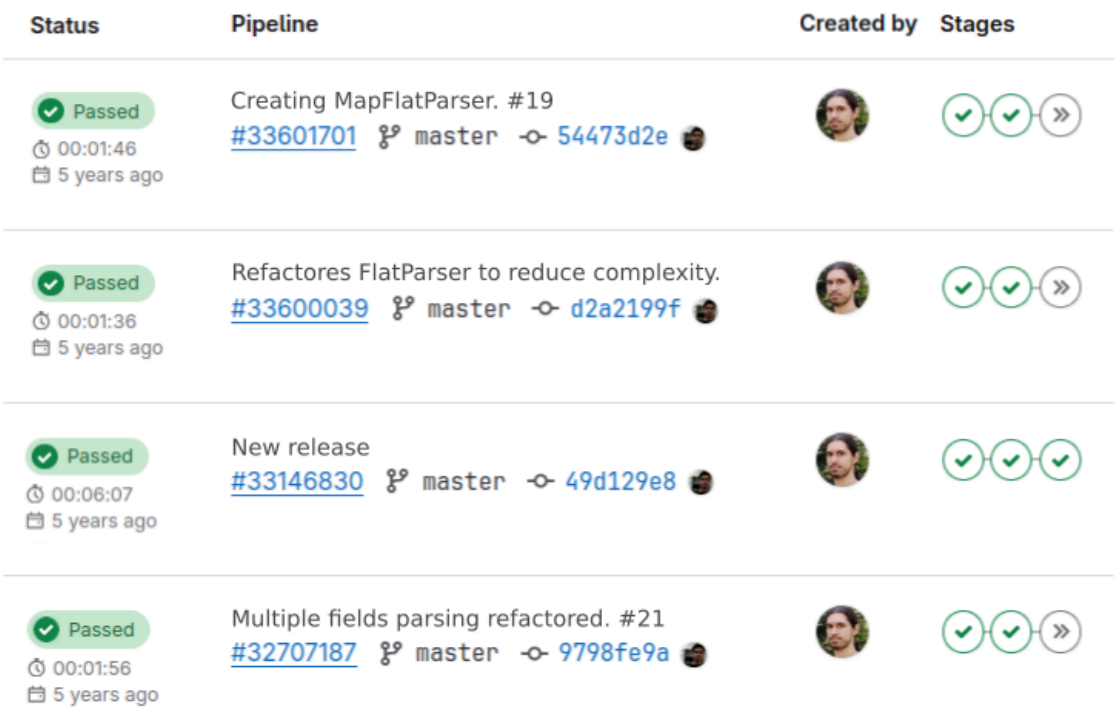
In the book *Continuous Delivery*, Jez Humble and David Farley (2010) argue for an *automated deployment pipeline*, in which any software version committed to the repository (that is, any *commit*) must be a candidate version for the production environment. In other words, the developer should not work with the idea that a certain version of the source code cannot go to production because a certain functionality is only half implemented. This is possible with the use of some additional techniques, mainly feature toggle, in which a new feature still under development is inactive in production. Once the commit is delivered to the repository, the value of that commit is put to the test by moving through the pipeline stages, such as the compilation and automated testing stages. Once the commit is approved in these early stages, the software can be sent to production by simply pressing a button in the pipeline. We call this process **continuous delivery**.

Commit

Commit is a technical term widely used in version control systems (Git, SVN, etc.) to identify the delivery of a version of the codebase to the source code repository. In other words, every time you want to make the code you just built visible to your colleagues, you make a commit. In short, commit corresponds to the "source code version". But be careful: the source code version does not necessarily correspond to the version of the running software.

A variant of the presented process is *continuous deployment*, a regime in which every version that successfully passes through the first stages of the pipeline is automatically sent to production. Many authors closely relate DevOps to continuous delivery and deployment, which is why we consider *Continuous Delivery* (HUMBLE; FARLEY, 2010) to be one of the seminal DevOps books. The following figure illustrates the idea of a deployment pipeline through a real example.

a)



b)

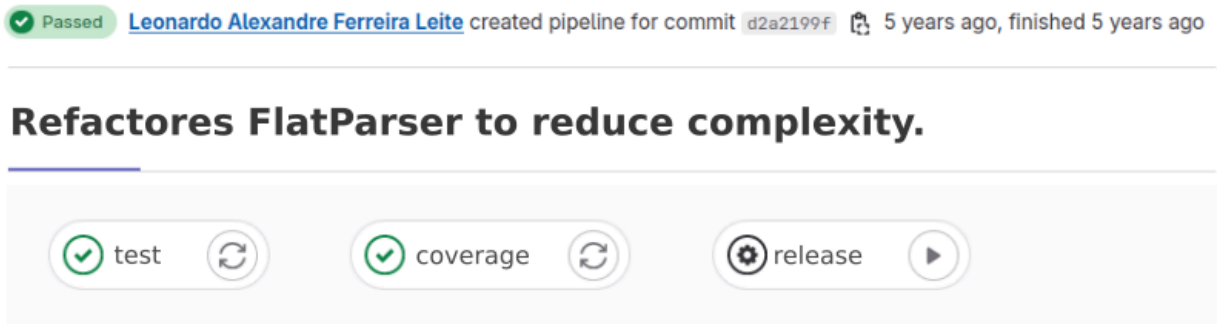


Figura 1.1: a) Each commit in the repository triggers the execution of an instance of the pipeline, which is composed of stages, such as the stage that runs automated tests; b) once the tests have passed, publishing the service (or a library, like the one in the example) is possible with the simple push of a button, which will trigger the publishing stage. The example was taken from the Fatiador project (https://gitlab.com/serpro/fatiador), which uses Gitlab CI as a pipeline system.

In addition to automating the delivery process, DevOps initiatives have also focused on using automated runtime monitoring to

improve software properties such as performance, scalability, availability, and resilience. From this perspective, the term "*site reliability engineer*" (BEYER, 2016) emerged on Google as a new term related to runtime DevOps work when concerned with engineering organizations and practices to ensure the reliability of large-scale systems. At Google, reliability engineers (SREs) handle operations, but they also invest at least 50% of their time to decrease the amount of toil required to keep services meeting service levels. In this way, after considerably reducing the operational overhead of the service, depending on the system in question, the developers themselves become responsible for the operation.

1.2 Our definition of DevOps

Given the history presented, we can now better contemplate the definition of DevOps proposed by the authors of this book:

"*DevOps is a collaborative and multidisciplinary effort within an organization to automate continuous delivery of new software versions, while guaranteeing their correctness and reliability*" (LEITE *et al.*, 2019).

While there is an industry trend toward adopting DevOps as a role, the "DevOps engineer," our definition is according to the original spirit of the DevOps movement, which was about breaking down the barriers between silos ("*DevOps is a collaborative and multidisciplinary effort within an organization*"). This spirit was portrayed, for example, in the aforementioned book *The Phoenix Project*. Note also that our definition does not restrict collaboration to just devs and ops, but suggests that it occurs between all departments in the company that can work together towards a cause, which is the ability to deliver software continuously ("*to automate continuous delivery of new software versions* "). In other

words, departments should not collaborate just because collaborating is nice, but because it is necessary given this great common objective.

By focusing our definition on continuous delivery, we are also in line with a common view in the literature that strongly associates DevOps with this concept, as evidenced by the title of one of the seminal articles on DevOps: *Why enterprises must adopt DevOps to enable continuous delivery* (HUMBLE; MOLESKY, 2011).

Our definition is also aligned with more recent developments that face the fact that it is not enough to deliver faster, but also that reliability is paramount ("*while guaranteeing their correctness and reliability*"), such as the discussion presented about SRE. Regarding "correctness", in the DevOps context, it is promoted by making the deployment of a version conditional on the successful execution of automated tests on that version.

Finally, we remember that the definition of DevOps is not consensual, and that, therefore, there are other valid definitions. But, with this definition as a basis, we hope you can already suspect incoherent definitions out there. One of the strangest ones we've seen, for example, was one that associates DevOps with Docker containers as opposed to using a *serverless* infrastructure.

SERVERLESS ARCHITECTURE

It is a model proposed by Amazon that abstracts the existence of servers (entities that host the application and may or may not be up and running). The model focuses on charging for the execution of business functions. A key constraint imposed by this architecture is the limit on the execution time of a function, which helps keep applications under this paradigm more scalable.

1.3 And who is the operator?

Before understanding the role of infrastructure operators or engineers, we need to reflect on the concept of *infrastructure*. In the context of IT, infrastructure refers to all the hardware, software, network, and facilities that are necessary to develop, test, deliver, manage, and operate IT services (ITIL FOUNDATION, 2019). It is also useful add to this definition the notion that infrastructure is a layer of software that can be provided as a commodity. In other words, although the infrastructure must meet the non-functional requirements of the application, it is normally provided regardless of the application domain, enabling organizations to offer it in a standardized way to different applications. We adopted this definition very carefully to include in infrastructure work not only physical machines but also virtualized machines and even deployment platforms like Kubernetes.

> **Kubernetes and Docker**
>
> Kubernetes (K8s) is open-source software used to automate the deployment, scaling, and management of containerized applications. Each container, using Docker technology, corresponds to a deployment of a specific service in an environment relatively isolated from the operating system hosting the container. It is common that, for reasons of scalability and availability, each service has several instances, thus being deployed in several containers. Learn more at https://kubernetes.io/ and https://www.docker.com/.

In this way, infrastructure engineers would be those who provide, with a certain uniformity, the infrastructure services on which the system depends (WOODS, 2016). The term "operations" refers to software operational activities, including software deployment, infrastructure preparation, and runtime operation of the software (monitoring and incident handling). In the pre-DevOps context,

operational activities were typically assigned to professionals taking care of the infrastructure. However, our research seeks precisely to understand the redistribution of these operational activities between infrastructure and development professionals considering the DevOps phenomenon. Therefore, instead of developers and operators, we prefer to talk about developers, infrastructure professionals, and operational activities.

And there is one more element to disregard the term "operator". In other industries, it is common to apply this term to workers who perform menial job to control machines. We would thus have engineers designing the machines and operators operating them. If the "machine" built by software engineers is the software system, operators can be the users of the system. It may sound strange to apply this term when the end users are customers, as in the case of email systems, for example. Still, the term seems well suited to situations in which users are employees of the customer (who demanded the system), as would be the case with supermarket cashiers. In this second case, the term "operator" (associated with the supermarket cashier) has clearly no relation to the "ops" of DevOps.

Still in this sense of "machine operator", we could consider that IT operators would be the workers who execute scripts to create environments and trigger restart commands when the system crashes. But infrastructure work is increasingly closer to the definition of "infrastructure engineer" than this vision of a more menial job. After all, providing infrastructure requires planning, sizing, and building automation. Additionally, when things are not working, investigative skills are required from infrastructure professionals. For all these reasons, although we sometimes use the term *ops* or operator, especially when commenting on already consolidated knowledge about DevOps, in this book we tend to prefer the expression *infrastructure engineers*, especially when discussing our research.

Finally, we remember that there is yet another important role in the scheme of keeping software operating, which is *customer support*. This role is sometimes also called operations and consists of non-IT-specialized labor responding to end-customer queries and problems. These professionals, although not technical, must be prepared to classify calls between business queries and technical problems, to redirect the second case to technical personnel.

1.4 DevOps Concepts

In this section, we present the fundamental DevOps concepts that emerged from our systematic literature review. This framework is composed of a map containing conceptual categories and four other maps containing concepts and relationships between these concepts.

The concepts are distributed into four categories:

1. Process
2. People
3. Delivery
4. Runtime

The **process** category encompasses concepts related to the business; **people** covers concepts and skills related to the culture of collaboration; **delivery** provides the concepts necessary for continuous delivery; and, finally, **runtime** compile the concepts necessary to guarantee stability and reliability of services in a continuous delivery environment.

While the *process* and *people* categories relate more to a management perspective, *runtime* and *delivery* relate more to an engineering perspective. Furthermore, although *delivery* concepts relate more to developers, *runtime* concepts relate more to the traditional operator role. The categories are represented in the following figure:

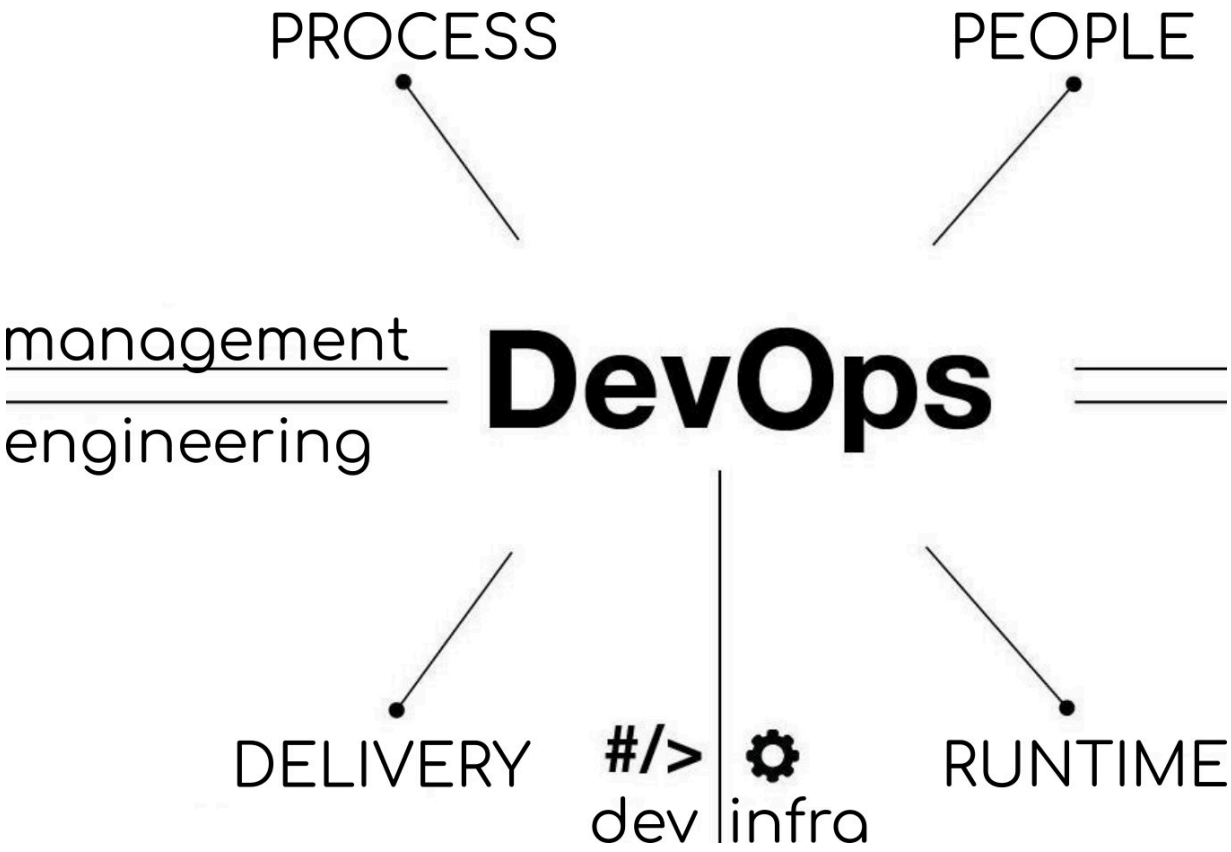


Figura 1.2: General map of DevOps concepts.

Before proceeding with the descriptions of the categories, we briefly present some reasons to support that these categories are sufficient to represent the DevOps world:

- It seems reasonable to separate the concepts about *devs* and *ops* as induced by the *delivery* and *runtime* categories;
- It makes sense to separate more technical concepts from less technical concepts, as caused by the separation between engineering and management perspectives;
- These categories correspond to our definition of DevOps: "a collaborative and multidisciplinary effort within an organization" in relation to *people*; "continuous delivery" is a process that is "automated" by *delivery* techniques, which also guarantee the correctness of the software; Finally, software "reliability" is promoted by *runtime* concepts.

1. Process

DevOps aims to achieve certain business outcomes, such as reducing risks and costs, complying with regulations, improving product quality, and increasing customer satisfaction. We group these concepts in the category of *process* concepts, presented in the following figure, which reveals that DevOps achieves such business results by carrying out a process with frequent and reliable deliveries. In particular, the diagram explains that continuous delivery leads to product quality and customer satisfaction due to the short feedback loop that is then promoted.

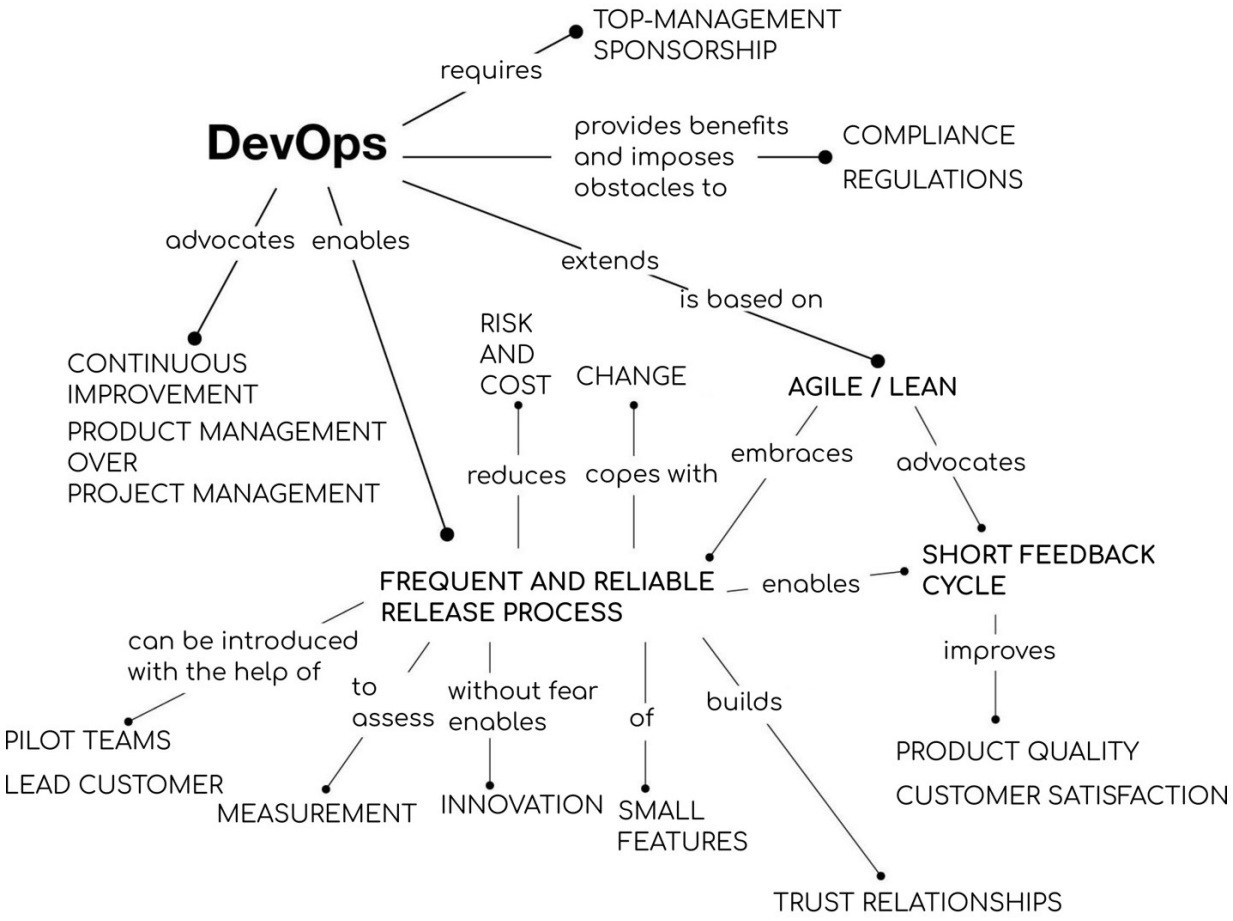


Figura 1.3: Map of DevOps concepts about process.

It can be argued that rigorous, hierarchical human approval processes can reduce risk, comply with regulations, and provide product quality. However, DevOps is different, as its practices are based on agile and lean principles, which embrace change and shorten the feedback cycle, as shown in the diagram above.

2. People

The term DevOps centers on the idea of bringing together development and operations *people* through a culture of collaboration. The concepts around this idea were grouped into the *people* category, which is shown in the following figure. It shows that DevOps aims to break down the walls between silos, aligning incentives across the organization.

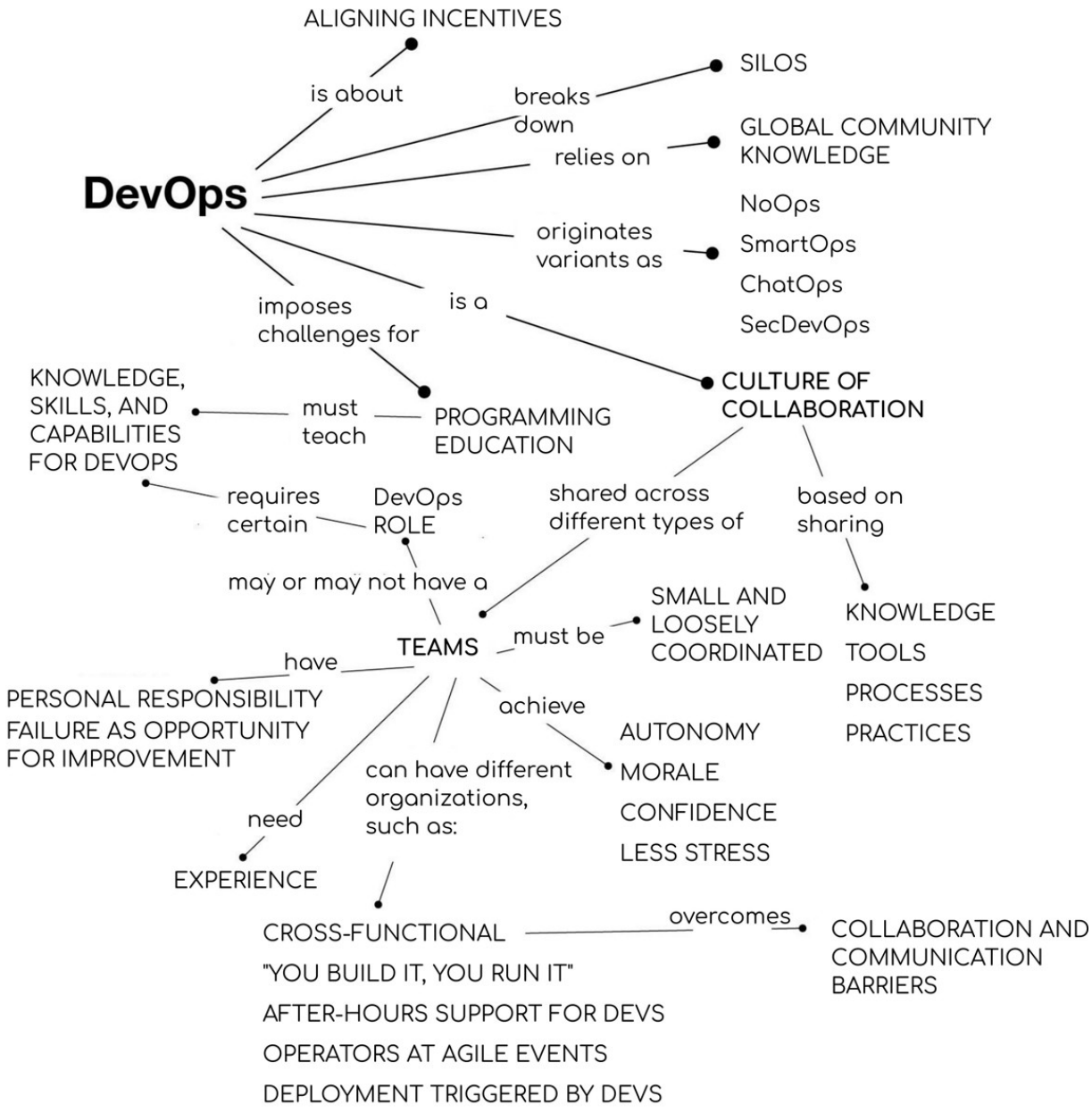


Figura 1.4: Map of DevOps concepts about people.

However, breaking down silos raises many questions about the concepts described in the diagram: how can organizations make a cultural change that implies more responsibilities for developers? How do developers acquire operational skills? Can developers and operators work on the same team and still hold different job titles? Should "DevOps" be a role? Should teams be cross-functional and include operators? Should an operator be exclusive to a team? What

does it mean to be an operator in the context of DevOps? Although concepts in this category are frequent in the literature, they are the ones that most point to open questions.

3. Delivery

The main strategy for achieving a frequent and reliable delivery process is the automation of the deployment pipeline, from which DevOps tools and techniques emerge. We group the concepts around the deployment pipeline in the *delivery* category, which is shown in the following figure.

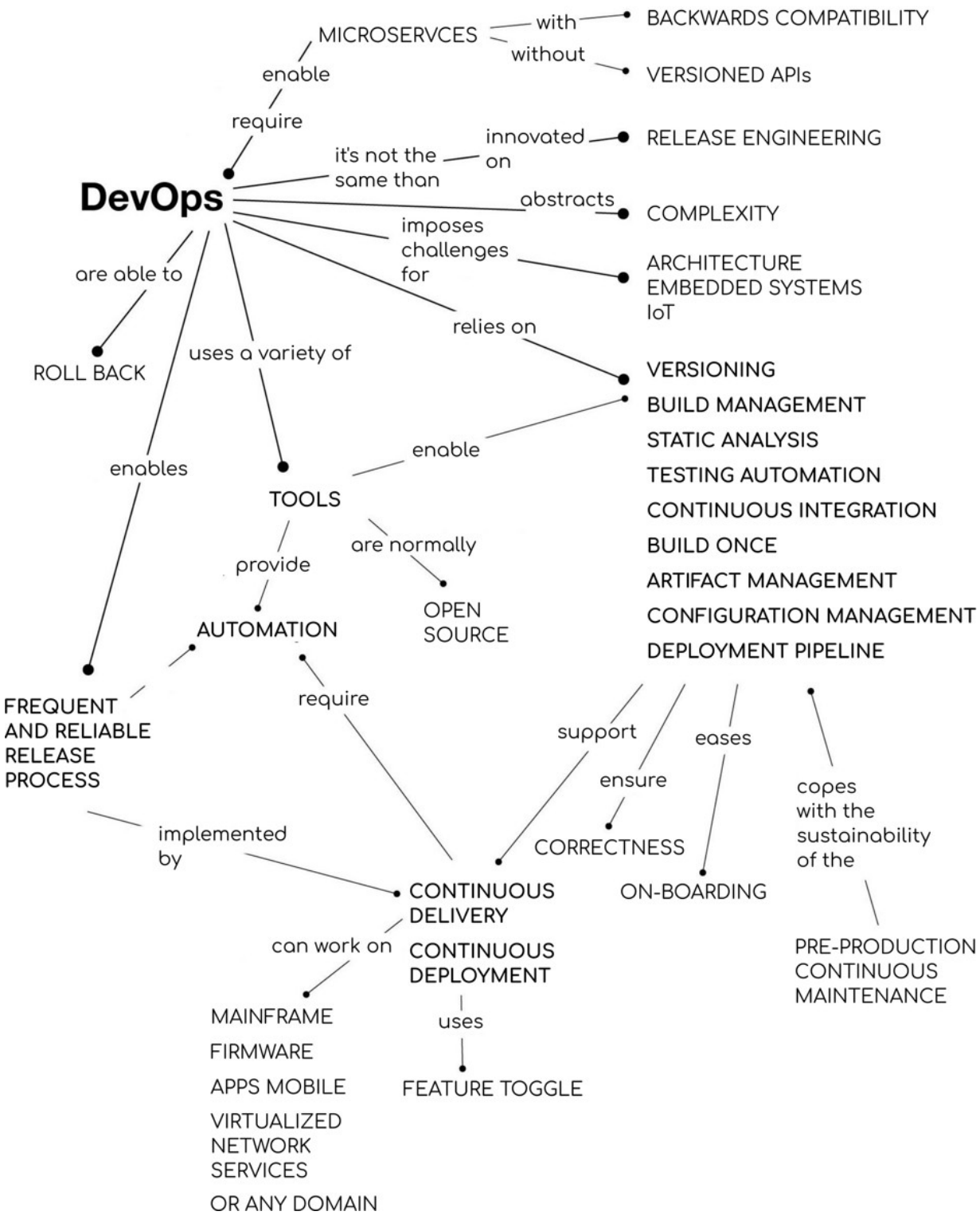


Figura 1.5: Map of DevOps concepts about delivery.

Automation tools, as illustrated in the diagram, are usully free / open-source software and enable the application of main DevOps practices, such as versioning, test automation, continuous integration, and configuration management. The diagram also shows the interconnection between DevOps and microservices: one supports the other, as we will explore later in the *DevOps and architecture* section of chapter *DevOps Challenges*. Other concepts related to microservices are also present, such as backward compatibility and API versioning.

There is still some debate about concrete tooling strategies, such as the containerized approach using Docker, on the one hand, and the convergence of continuous configuration, as in Chef and Puppet, on the other. We discuss these issues in the tools section. However, we consider that the concepts of *delivery* are much more stable and accepted by the community than the concepts of *people*.

4. Runtime

It is not enough to continually deliver new versions, it is also necessary that each new version is stable and reliable. Thus, *runtime* concepts are a consequential and necessary extension of DevOps. The *runtime* category of concepts is presented in the following figure, which displays desired properties for software at runtime, such as performance, availability, scalability, resilience, and reliability. The figure also shows paths to achieve the mentioned properties, such as using infrastructure as code, virtualization, containerization, cloud services, and monitoring.

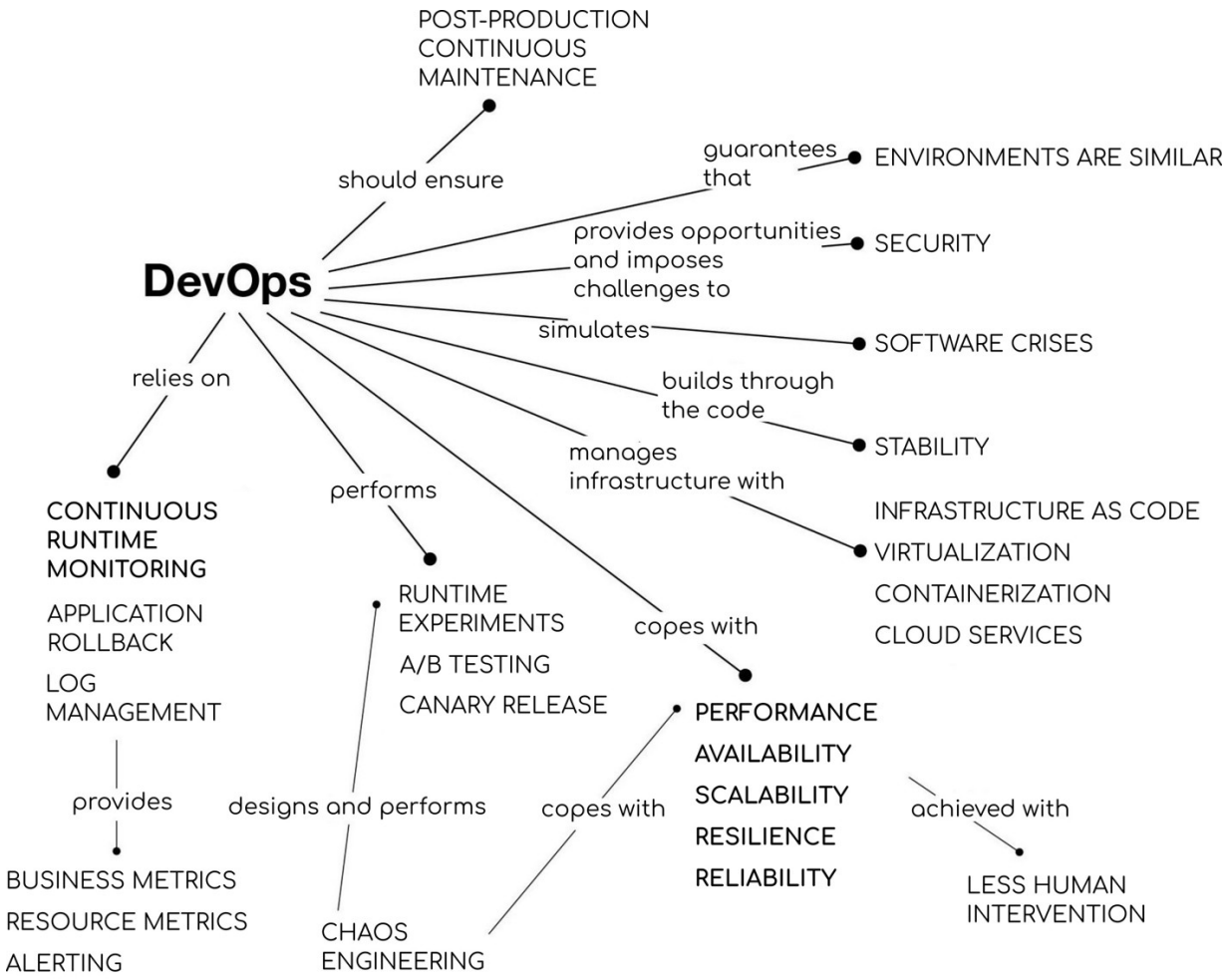


Figura 1.6: Map of DevOps concepts about runtime.

As illustrated by the diagram, DevOps can monitor high-level business metrics (e.g., sales volume) or low-level resource metrics (memory, CPU, etc.). Another topic, also presented in the figure, is the execution of experiments in the production environment, such as fault injection to ensure software reliability, as advocated by the chaos engineering approach (BASIRI, 2016). All of this reduces human intervention to ensure software reliability, which is another factor that challenges the traditional role of operators, one of the issues posed by concepts about *people*.

Group exercises: DevOps journey retrospective

The conceptual maps presented in this section can be used for guiding a retrospective of a team's DevOps journey.

Each turn is defined by the combination of a team member and one of the four large concept maps (process, people, delivery, and runtime). People and maps alternate sequentially. On their turn, the team member chooses a concept from the current map. All team members raise one of the following cards:

- We made it and we're happy about it.
- We made it, but it doesn't seem to matter much.
- We couldn’t, but we want to!
- We couldn’t, but we don't see the value in it.

Then people discuss the assessment, and turns continue until the predefined time limit (e.g., one hour).

In the next chapter, we will address the impacts that the DevOps movement has brought to both technical personnel and management.

Capítulo 2
Impacts of DevOps

DevOps principles, practices, and tools have already changed the software industry. However, many professionals, both engineers, and managers, are still not aware of how their daily work can be affected by these principles, practices, and tools. When analyzing studies on DevOps, we found several implications of adopting DevOps for industry professionals. In this section, we present these implications to help professionals adapt to the most important impacts of DevOps in their areas. We split the impacts into two sections: impacts for the technical staff (developers, engineers, analysts, etc.) and impacts for the management staff (managers, bosses, executives, etc.). Finally, these implications raise some considerations that lead us to some major DevOps challenges, which will be delved into in depth in the next chapter.

2.1 Impacts on the technical staff

Based on the reviewed literature, we now list the implications of DevOps that affect how technical staff should design systems, interact with their partners, and even adopt certain processes, such as incident handling.

Microservices: Adopting microservices architecture is recommended in conjunction with adopting continuous delivery. A loosely coupled and well-encapsulated microservices architecture leads to good system testability and deployability. As microservices come with new challenges, and we will discuss more about this topic in the next chapter.

Cloud services: Architectural patterns involving cloud services can assist in the deployment and operation of applications, reducing the need for dedicated operations teams.

Reverting versions: Some authors claim that DevOps engineers should be able to rollback an application to a previous version in case of problems after deployment. However, in scenarios involving complex integrations or database evolution, reverting the version of a service may be an unfeasible task. In other words, if the service has evolved to keep up with the evolution of a dependency and you can't rollback the version of the dependency, you won't be able to roll back the version of the dependent service. Alternatives to trying to avoid rollback are the use of feature toggles or addressing the root cause of the problematic delivery.

Embedded systems and IoT devices: Deployment on embedded systems can be difficult, especially when only the organization contracting for software development owns and controls the hardware platform. In this case, engineers must adapt delivery mechanisms according to the specificities of embedded systems and IoT devices.

High-Frequency Delivery Inhibitors: Engineers must be aware of scenarios that can impose barriers to high-frequency delivery. Updating automation control systems software may require interrupting a factory's production, which can be costly. Traditional techniques to prevent the application from going offline during deployment, such as *Blue-Green Deployment* and *Canary Release*, may not be suitable for critical applications such as medical equipment and factories. The mobile application scenario may be less critical, but it also suffers from delays in new deliveries, since the mere publication of a new version of an app in one of the distribution stores, such as the Apple Store, can take more than a day.

Blue-Green Deployment and Canary Release

Blue-Green Deployment and *Canary Release* are techniques to mitigate risk when delivering new software versions to the production environment.

In *Blue-Green Deployment*, an environment B is built with the new version of the system, but with an infrastructure similar to the current production environment A. The requests are then switched to the new production environment B, and the launch is done. However, if an apparent problem arises, it is possible to quickly rollback the system by redirecting requests to environment A, which will become inactive (FOWLER, 2010).

In *Canary Release*, the new version of the software coexists for a while with the previous version, so that the percentage of requests that are directed to the new version is gradually increased, as the absence of errors provides more confidence in the new version (SATO, 2014).

Tests: Although companies recognize the importance of automated tests, they still have difficulty exploiting them to their full potential. This is especially true in the context of UI test automation. One factor that makes test automation difficult is the availability of hardware for load testing: ideally the load testing environment should have the same resources as the production environment, which can be very costly. Another factor is the assessment of user experience: did changes to an existing functionality worsen the user experience? This is hardly automatable at a pre-production stage. Other recommendations that are difficult to follow are the parallelization of test suites to reduce test execution time (that is, running more than one test at the same time to save time) and the elimination of tests that seem to fail randomly, the so-called *flaky tests*.

Quality Assurance (QA) Team: Quality assurance skills are required to find specific error scenarios and corner cases. However, keeping the quality assurance team separate from the development team is questionable. Agile and DevOps practices require a change in the role of quality assurance teams or even their elimination.

Legacy Systems: Although it takes a lot of effort to achieve continuous delivery on mainframe platforms, there are reports of success. Some legacy architectures may not have been designed with automated test execution in mind. Still, teams should be aware that cultural factors, such as managers who say "this is how we've always done it here," may limit the adoption of continuous delivery more than technical factors.

Learning: software professionals must be prepared to learn new tools, with a focus on automation. These people must also deal with the constant change of these tools, needing to maintain heterogeneous environments and frequently deal with technological migrations.

Building the deployment pipeline: The benefits provided by a deployment pipeline are many. However, engineering must be aware that configuring the infrastructure for continuous delivery can require considerable effort. Dividing the system into microservices also requires building several pipelines. Engineers should not try to build the entire continuous delivery ecosystem in a single step: an approach based on continuous improvement is preferable.

Pipeline maintenance: running the pipeline generates many artifacts, such as builds and logs. Artifacts such as production logs, bug reports, parameter configuration, and temporary files must be properly archived and removed at some point. Despite its importance for the sustainability of the pipeline, organizations often neglect this maintenance process.

New Team Member Onboarding: Automation built for continuous delivery can also promote faster onboarding of new team members.

This is about applying all DevOps expertise not in the production environment, but in the local environment of the developer. Tools like Docker are great allies here.

Incident handling: Software developers must be trained in software security and must cooperate with incident handling, tracking bugs and system failures, or else must assume primary responsibility for these activities.

Coding for stability and security: While software stability and security are traditional concerns of operations personnel, in a DevOps context, these non-functional requirements must be leveraged by software implementation, including here both deployment automation code as for the application code, which considers the possibility of failures and delays in the underlying infrastructure.

2.2 Impacts for managers

We now list the implications of DevOps related to how managers should view the DevOps phenomenon: the necessary managerial and cultural paradigms, the training of people, the structuring and evaluation of the DevOps adoption process, as well as the expected results of this process.

Adoption of *lean* principles: As DevOps is based on *lean* principles, organizations wishing to adopt DevOps must also master these principles. In particular, Kim (2012) recommends: (1) mapping the value stream to optimize global system performance, rather than seeking local optimization; (2) amplifying continuous feedback loops to support necessary corrections; and (3) improving daily work through a culture that promotes frequent experimentation, risk-taking, and learning from mistakes, knowing that practice and repetition are prerequisites for mastery.

Local optimization vs. global optimization

Imagine an executive pressuring the development team to increase their productivity. Imagine that, at great cost, the team managed to double its productivity (that is, what they previously developed in a month, they now do in two weeks). Excellent, right? Well, it depends... Consider that, after the development phase, the infrastructure takes a week to provision the environment and approval, that the customer takes two months to approve, and that, again, the infrastructure takes another week to provision the production environment.

In this situation, the productivity improvement (in terms of the product reaching the end user) will be only 17%, not 100%. In other words, it is not enough for everyone to take care of their part. It is necessary to analyze the whole and prioritize the optimization of process bottlenecks. In this example, perhaps it would be better to invest in ways to speed up approval: improving manuals, promoting assisted approval, etc. This idea, also known as the *Theory of Constraints*, is illustrated in narrative form in the book *The Goal*, by Eliyahu Goldratt — a read that is very worth reading!

DevOps Adoption: *How* to implement DevOps in an organization is a critical question for managers. However, the lack of a consensual definition of DevOps is one reason that makes this decision difficult. In addition to having already provided a definition of DevOps (chapter *What is DevOps?*), this "how to deploy" relates to the main question of this book, which will be covered in depth in its second part.

Assessment: An organization must be able to measure the success of the DevOps adoption process. Still, any top-down imposition of metrics-based assessment should be used with great caution. If personal evaluation depends on these metrics, engineers may focus

on simply producing convenient numbers rather than actually improving the software process. Furthermore, it is necessary to consider the risks of local optimization, as explained in the previous box. We will further explore this issue in the next chapter, on the *DevOps Challenges*.

Training: DevOps requires additional technical skills from software professionals. Developers must acquire skills common to operations personnel, and vice versa. It is necessary to carry out training in the use not only of tools but also in the use of DevOps concepts so that professionals can keep up with the rapid evolution of tools. Top management support for training is often necessary in many contexts.

Roles: The definition of roles affects the hiring process and the design of training. While the DevOps movement emerged to bring developers and operations people together, today the industry embraces the "DevOps engineer" role, who performs tasks primarily related to script automation, continuous integration, and continuous delivery practices. However, this role is confused with others, such as the release engineer and the build engineer. There is no consensus in industry and academia on the definition of these roles. When analyzing hiring announcements, some researchers have found that DevOps positions generally do not entail build and release management duties, while other researchers have identified that these three roles (DevOps, release, and build engineers) share common tasks. The challenge for managers to define roles is also related to how to structure development and infrastructure teams, as we will discuss in depth later (the topic of Part 2 of this book).

Culture: Jez Humble (2017) has already stated that, typically, the obstacle to adopting continuous delivery is not the individual skill level of employees, but rather the set of failures in management and leadership. High-performance organizations strive, at all levels, for continuous improvement, and do not treat workers as replaceable "resources" who must simply perform tasks as efficiently as possible. Top and low-level management is responsible for creating an

environment in which failures are permitted and in which people constantly seek to improve. We will go a little deeper into the discussion about culture in the chapter *Organizational Structures Beyond DevOps*.

Increase in delivery rate: Researchers have already reported an increase in delivery rate per developer and an improvement in the predictability of the defect rate with the adoption of continuous delivery. Researchers from the University of Brasília (UnB) reported how, after investments in DevOps and continuous delivery, the number of deliveries per semester in a project did not change even after a considerable reduction in the number of team members (SIQUEIRA *et al.*, 2018).

Building trust: When an organization builds software for another large organization, especially for the government, it is common for the relationship between contractor and contractor to be dominated by distrust, which leads to development processes with too much bureaucracy. However, continuous delivery promotes a relationship of trust between contractor and contractor in software development, even in government projects, as was observed in a successful partnership project between the government and the university (SIQUEIRA *et al.*, 2018).

Building for government: Although building software for the government may involve more bureaucratic processes, requirements and prioritization can often change due to political reasons (SIQUEIRA *et al.*, 2018), which leads to the need to shorten the release cycle through agile and DevOps practices in the government setting.

Let us now analyze in more detail some implications that we consider more challenging.

CAPÍTULO 3
DevOps Challenges

We have already seen that the analysis of the concepts and implications of DevOps raises some questions to be debated. Based on the literature, we will now discuss in this chapter some of the DevOps challenges faced by management and technical personnel that are not fully addressed by the current state of the art.

3.1 DevOps and architecture

Highly coupled monolithic architectures (that is when a single deployment delivers all modules of a system into production) are obstacles to effective continuous delivery. Managing the complex network of dependencies between components and between teams impacts the deployment pipeline. A typical problem is the perception of increased risk that deploying one component could damage or make other components in the same monolith unavailable. Another issue is when the deployment of a monolith, hosting many components, necessarily implies momentary unavailability, which may be undesirable. These issues end up generating friction between teams, as the team responsible for one component wishes to block a deployment desired by the team from another component.

Therefore, an essential principle for successfully adhering to continuous delivery is the adoption of an architecture composed of small, independently deployable units, also called **microservices** (LEWIS; FOWLER, 2014). In such an architectural style, services interact across the network and are designed around business abstractions.

While some authors say that microservices facilitate effective DevOps implementation, others say that microservices require DevOps, as deployment automation reduces the overhead of managing many microservices. Designing an architecture with loosely coupled and well-encapsulated microservices ensures two architectural attributes necessary for continuous delivery: *testability* and *deployability*. The existence of a test suite per microservice reduces the wait for deployment since running a unified battery of tests from a monolith tends to take too long.

The adoption of microservices also brings challenges. First, the application of some operational patterns varies greatly, such as startup scripts, configuration files, administration endpoints and ways of accessing logs. This heterogeneity can be a barrier to productivity for newcomers to a team (or even for veterans dealing with different microservices following different patterns).

Second, microservices must be deployed to production with the same set of versions used in integration tests. These challenges can be overcome by adopting minimal standardization of microservices across the organization (of course, great care is needed against excessive standardization). Standardizing just the *health check* URL can be a good start.

HEALTH CHECK

The *health check* URL is an administrative endpoint that exposes information about the health of a microservice, mainly indicating whether the service is available or not. Consider as an example a reservation system ("yourhotel.com") with a search microservice; in this context, the health URL could be `https://yourhotel.com/search/health` .

Some microservices management patterns associated with DevOps are:

- A deployment pipeline per microservice: after all, each microservice must be, by definition, deployable independently of other microservices.
- Log aggregator: important when a microservice has several instances (a common situation), so the developer do not need to guess in which instance the error under investigation occurred.
- Service registry: in which a developer can search for microservices available in the organization.
- Correlation IDs (also called trace IDs): microservice A calls microservice B, which calls microservice C, where an error occurs. How to combine error logs from different microservices for better diagnosis? The trick is that each invocation passes a common correlation ID, so that error records are associated with this ID.
- Segregation of source code, application configuration, and environment specification: the application source code does not store anything that varies with the environment (example: database address), nor the definition of the environment itself (example: host operating system customizations).

Other complementary patterns are:

- Strangler application: new microservices are born and gradually reduce the use of the monolith.
- Load balancing: requests to a microservice must be distributed among the instances of that service.
- Consumer-oriented contracts: greater focus on testing the service with its expected uses than on exposing customers to a specification of everything the service is capable of doing.
- Circuit breaker: in case of many requests in the environment, it may be healthy to interrupt the flow of requests so that the services can recover.
- Backward compatibility: a new version of a microservice must continue to be functional for the uses already made by its customers, otherwise a lot of conversation and coordination will

be needed for a deployment that breaks this backward compatibility.
- Versioned APIs: some people assign a new version number, explicit in the URL, for each release of a microservice. However, this point is controversial, and some communities generally do not recommend this type of versioning.

In highly regulated environments, the appropriate use of microservices can limit the impact of restrictive regulations to just a few specific modules and people in the system. Furthermore, the use of a Platform as a Service (PaaS) can automate much of the compliance checking – a model used, for example, by the United States federal government (HUMBLE, 2017). The PaaS model is also recommended for its operational simplicity, as is the case with other cloud services: storage services, asynchronous processing with queues, email delivery, and real-time user monitoring.

Finally, although reusability is a historical objective of software engineering, there are researchers who recommend that engineers not focus so much on this. Reuse brings coupling, which can be a major bottleneck for continuous delivery. So, each team must consider the balance between reuse and independence from other components, services, and teams.

3.2 How to measure DevOps adoption?

Some researchers have proposed maturity models for DevOps adoption and continuous delivery. In one article, for example, the authors provide models to help organizations measure their current DevOps maturity level and determine areas lacking investment (FEIJTER *et al.*, 2018). However, none of these maturity models are widely adopted by the industry. In fact, most organizations have almost no visibility or reliable measurement of their software delivery practices.

In an academic article, Nicole Forsgren and Mik Kersten (2018) state that both survey data and system data should be used to measure DevOps transformations. System data can provide a continuous flow of information, although configuring the aggregation of metrics from different sources can be challenging. Survey data, provided by humans, provides a holistic view of issues outside the system, such as culture, job satisfaction, and burnout, and can even point to problems in system data collection.

Some examples of metrics based on survey data used in the *State of DevOps Report* (FORSGREN *et al.*, 2018) are: delivery performance (consisting of time from commit to production, deployment frequency, and recovery time); time spent on unplanned work and rework; typology of organizational culture (pathological, bureaucratic or generative) based on climate for learning, job satisfaction, developers and operators with win-win relationships, use of version control and automated testing; employee engagement based on the *Net Promoter Score*, indicating the likelihood that employees will recommend the company's products and services (another interesting use of the *Net Promoter Score* is to evaluate how much employees indicate their companies or teams as a good place to work).

Metrics to evaluate DevOps in a specific company were used by researchers Barry Snyder and Bill Curtis (2018). Metrics such as productivity, defect rate, money spent, cycle time, and the number of builds were aggregated into a "total quality index". Most importantly, the aggregation of data from different silos was used to align organizational efforts. The authors observed a greater increase in the total quality score of teams adopting agile and DevOps practices, which helped executives justify investments in agile-DevOps transformation.

There is a great lack of industry consensus on how to perform measurements on the processes involved in software production. Furthermore, designing a good survey requires some experience (it's very easy to design bad questions without realizing it). Thus,

academic researchers could provide guidelines for the collection and analysis of metrics for evaluating the software delivery process, as done and reported by the authors of the article *DevOps Competencies and maturity for Software Producing Organizations* (FEIJTER *et al.*, 2018). In the health area, this model is very mature, in which researchers empower therapists with standard protocols for diagnostic purposes. However, even if you have these templates, you can follow Forsgren and Kersten's (2018) guidelines for applying forms: they should be limited to 25 minutes and applied no more than every 4 months. Forsgren and Kersten also advise avoiding sending the forms to managers and executives, as this tends to overestimate the maturity of their organizations.

3.3 Teaching and learning DevOps

We have more than 100 DevOps tools available out there, all of them constantly evolving. Professionals must know how to choose which ones to study in depth, which tools to have only some familiarity with, which ones to try, and which ones to ignore. Such choices must be based on a conceptual analysis of personal and organizational demands and perspectives. Our concept maps (chapter *What is DevOps?*) and our table of top DevOps tools (chapter *DevOps Tools*) can guide technical people to prioritize improving their professional skills.

Caution is needed when managers assume their employees lack competence. When asked, "*Where did you get these amazing employees?", a Netflix architect responded, "_I got them from you!*" (HUMBLE, 2017). The lesson here is that a culture of autonomy and responsibility is an important motivating factor for the continuous learning of employees.

In software engineering courses, operational skills, often neglected in University education, should now be mandatory. However,

designing good DevOps courses is challenging. Exercises in which students build distributed systems with adequate availability, scalability, and performance require not only some infrastructure but also automatic corrections. While good auto-grading platforms exist in the context of coding assignments, evaluating infrastructure code and non-functional requirements is more difficult to automate. Teachers have been trying to automate some parts of executing and evaluating submissions. One of the reports demonstrates the difficulty in fully automating the correction of exercises, a non-trivial task (CHRISTENSEN, 2016). In another case, the automation of the generation of evaluation reports was based only on the static evaluation of artifacts such as source codes, tasks, commits, branches and tests (BAI *et al.*, 2018).

Knowledge in software engineering is always evolving, and any software engineer must also have self-learning skills to overcome everyday challenges. Social interactions on internet forums are a vital source of information in the daily activities of engineers. Believing that this can be improved, researchers built a social network to help DevOps engineers analyze results of past deployment executions as well as communicate and exchange ideas to better understand the trade-offs involved in the space of deployment solutions (MAGOUTIS *et al.*, 2015). This work shows evidence that more can be done to support engineers' self-learning in their daily activities.

We will now complete our overview of DevOps (Part 1 of this book) with the more practical side of DevOps, that is about DevOps tools and actors and their relationships with concepts.

Capítulo 4
DevOps Tools

Tools matter, but caution is required to avoid turning them into the core of a DevOps strategy. Adopting new tools requires significant effort, and applying that effort without considering the expected result can lead to considerable waste. Furthermore, it is impossible for one person, or even a team, to master all available options.

Although much has already been said about the topic, the focus of our presentation is the association between tool categories and DevOps concepts. This association empowers professionals to make better-informed decisions about adopting these tools. This conceptual approach becomes even more relevant when considering the large number of tools available and the constantly changing DevOps tools landscape.

We consider **DevOps tools** to be those pursuing one of the following objectives: (1) assist human collaboration between different departments; (2) enable continuous delivery; or (3) maintain software reliability (these objectives are derived from our definition of DevOps, presented in the *What is DevOps?* chapter).

We then present a set of categories of DevOps tools and associate for each of these categories: *(i)* examples of popular tools; *(ii)* target users — devs (people responsible for development) or ops (people responsible for operations and/or infrastructure); *(iii)* objectives (human collaboration, continuous delivery or reliability); and *(iv)* DevOps concepts.

We hope that our approach to DevOps tools can help professionals with the following points: *(i)* encourage a focus on the organization's requirements when choosing the most appropriate set of tools; *(ii)* contribute with a critical view of the objective of each tool, and *(iii)*

show how the use of certain tools contributed to blurring the boundaries between development and operations.

The categories of tools, discussed in the next sections and presented in the following table are: knowledge sharing, source code management, build process, continuous integration, deployment automation, and monitoring & logging.

Category	Examples	Actors	Objectives	Conc
Knowledge sharing	RocketChat GitLab wiki Redmine Trello Notion	Everyone	Human collaboration	Collabora culture Knowled sharing Break do silos Collabora between departm
Source Code Management	Git SVN CVS ClearCase	Dev/Ops	Human collaboration Continuous delivery	Versionir Collabora culture Knowled sharing br>Breal silos Collabora between departm
Build process	Maven Gradle Rake npm JUnit Sonar	Dev	Continuous delivery	Delivery engineer Continuo delivery Automat Test auto

Category	Examples	Actors	Objectives	Conc
				Correctn Static an
Continuous integration	Jenkins GitLab CI Travis Nexus	Dev/Ops	Continuous delivery	Frequent reliable c process Delivery engineer Deploym pipeline Continuo delivery Automat Artifact manager
Deployment automation	Chef Puppet Docker Terraform Kubernetes Heroku Open Stack Rancher FlyWay	Dev/Ops	Continuous delivery Confiability	Frequent reliable c process Delivery engineer Configur manager Continuo delivery Infrastru as code Virtualiz Containe Cloud se Automat
Monitoring & logging	Nagios Zabbix Prometheus	Ops/Dev	Reliability	"You buil run" After-hou

Category	Examples	Actors	Objectives	Conc
	Grafana Jaegar Logstash Graylog			support Continuo runtime monitori Performa Availabili Scalabilit Resilienc Reliabilit Automat Metrics Alerts Experime Log manager Security

4.1 Tools for knowledge sharing

DevOps strategy focuses on human collaboration across silos. This implies different departments sharing knowledge, processes, and practices, which requires the common use of certain tools. We cite GitLab and Rocket Chat as examples of tools in this category.

GitLab not only offers a code repository but also has a *wiki* system that enables developers and operators to share knowledge. GitLab offers an issue management system, which is typically used as a focal point by developers and business analysts. Making operations people familiar with the issues tracker also helps them to understand the products and projects contexts. Furthermore, documenting production problems in the issue management system is an essential

step in dealing with the prioritization of non-functional requirements in the development flow.

ChatOps (Chat and Operations) is a model that connects people, tools, processes, and automation through tool-mediated conversations. Rocket Chat is a communication tool that implements ChatOps, enabling a greater degree of automation and tool integration through the use of *webhooks* and *chatbots*.

These tools support *human collaboration* and the *people* category of DevOps concepts. However, despite the importance of human collaboration in DevOps strategy, the *DevOps Periodic Table*, a well-known list of over 100 DevOps tools (https://digital.ai/learn/devops-periodic-table/), includes just 8 tools for collaboration. This fact suggests that the concepts of human collaboration and concepts of *people* were eclipsed in the DevOps movement by more technical goals and concepts so that the community ended up focusing much more on deployment automation tools.

4.2 Tools for source code management

Source code management tools often aim to promote collaboration between developers. They are basic blocks for implementing continuous integration and therefore continuous delivery. From this traditional perspective, one could relate source code management exclusively to the *delivery* category of DevOps concepts. However, source code management can also be used by infrastructure personnel to store artifacts, automate scripts, and access system information that affects operational activities.

For example, a development bug could cause a memory leak, affecting operations. By storing infrastructure-related artifacts, whether in infrastructure-as-code style or simply in textual form, the infrastructure area provides developers with better insight into how

the software runs. Therefore, sharing source code between areas becomes a point of collaboration between these parties.

The most traditional tools in this category are SVN and Git. More complete platforms, such as GitLab and GitHub, can also wrap Git, providing easier-to-use visualizations for code changes, as well as integrating with additional tools like the issue manager.

4.3 Tools for the build process

The *build* tools (construction of deployable units) are highly centered on the developer. Their objectives are about enabling continuous delivery, therefore they are associated with concepts in the *delivery* category of DevOps concepts. We have considered not only tools that generate deployable packages, but also called *builds*, but also tools that generate essential artifacts and feedback using the source code as input.

Each programming language has its build tools supporting dependencies resolution, implementation of custom tasks, or generating deployable packages. Examples of dependency managers, also called package managers, are Pip for Python, RubyGems for Ruby, NuGet for .NET, npm for JavaScript, and Maven and Gradle, which compete in the Java landscape. Gradle eases the implementation ofcustom tasks during compilation, as it is done by GNU Make, often used for GNU/Linux packages, or Rake for Ruby. Some of the tools mentioned, such as Maven, are also responsible for producing the deployable package, such as WAR files for Java environments. However, for some languages, such as Python and Ruby, there is no need to produce a deployable package as a single-file artifact. It is also possible to use packaging tools applicable to any technology but coupled to the target environment, such as Debian packages.

Each programming language also has some unit testing frameworks that provide vital feedback to developers about the correctness of the software. Some examples are JUnit and Mockito for Java, RSpec for Ruby, NUnit for .NET, PyTest for Python, and Jest for JavaScript. More sophisticated testing automating user actions for web applications is possible with navigation automation tools such as Selenium, Cypress, and BrowserStack. Furthermore, many of Selenium's features are also possible through the *headless* mode of browsers like Chrome and Firefox (when the browser works in the background without having to load the visual part).

Another type of feedback for developers is regarding source code quality, which is provided by static source code analysis tools such as SonarQube and Analizo. SonarQube classifies code issues and evaluates coverage metrics as well as technical debt items across multiple programming languages through its plugins. Analizo (www.analizo.org) supports extracting a variety of source code metrics for C, C++, C#, and Java. In general, source code analysis tools can point to issues in the source code, such as style nonconformity, maintainability issues, risk of bugs, and even vulnerabilities. This type of tool can be provided as a service, such as Code Climate, or even within the developer environment through tools such as PMD for the Eclipse IDE. There are also plugins similar to PMD for the VSCode and IntelliJ IDEs.

As a service

The expression "as a service" means that a certain product can be used online without anyone having to install the system for a specific user or set of users. Example: a company can have its developers using Gitlab.com without needing an infrastructure team to install and manage the GitLab software on the company's infrastructure.

A fundamental requirement for a build tool in the DevOps context is automation. There are some products whose build actions are triggered only through clicks on a Graphical User Interface (GUI), which is not acceptable in a continuous delivery process.

4.4 Tools for Continuous Integration

Continuous integration tools orchestrate several automated actions that together implement the deployment pipeline. The following figure provides an example of an idealized pipeline relating the pipeline steps to types of DevOps tools. Among the steps orchestrated by the pipeline are the generation of packages, execution of automated tests to verify correctness, and deployment in different environments (both production and non-production). These tools are related to concepts in the *delivery* category of DevOps concepts.

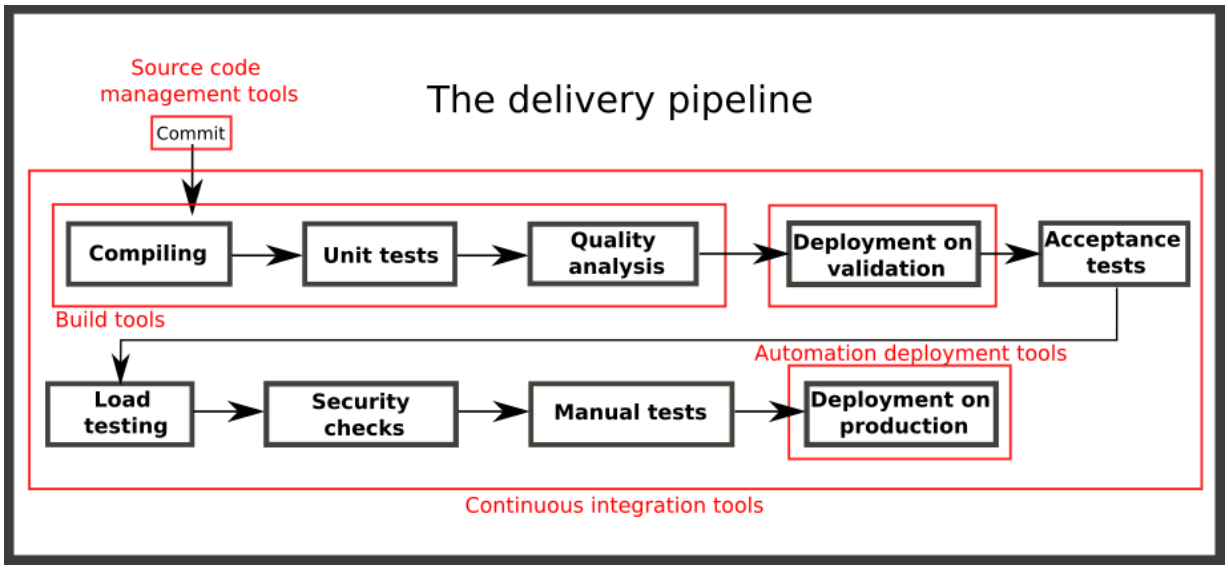


Figura 4.1: Continuous integration tools integrate with source code management tools so that at each commit, tasks are performed by build tools and automated deployment tools. Note: monitoring & logging tools come into play after production deployment, while knowledge-sharing tools span the entire delivery cycle.

The main actors responsible for defining the structure of the pipeline are typically developers. Infra staff generally collaborate in defining the implementation steps; they are also responsible for keeping the continuous integration infrastructure working as a service for developers. For this, infrastructure professionals use continuous integration tools, such as Jenkins, GitLab CI, or GitHub Actions. Deployable packages can be stored in repositories such as Nexus to later enable rollback (reverting the application to a previous version).

Because continuous integration infrastructure downtime results in disruption to continuous delivery, it is also common to use highly available third-party services such as GitLab.com (GitLab as a Service) and Travis.

4.5 Tools for deployment automation

Deployment automation tools are employed in the pipeline deployment stages to make the continuous delivery process viable. They enable frequent and reliable deployment processes, as well as the realization of other concepts in the *delivery* category of DevOps concepts. While using tools for deployment automation is a joint effort between devs and ops, the core mission of continuous delivery puts deployment processes under the control of the business.

Every automated deployment approach is based on the concept of "infrastructure as code." This requires engineers to specify servers, networks, and other infrastructure elements in files modeled as software code. In this model, deployment and infrastructure definitions are shared between developers and infrastructure personnel, allowing cooperation between them.

Automated deployment can encompass not only deploying the package to the target environment but also provisioning that same target environment. Such provisioning is generally performed in

cloud environments such as Amazon Web Services, Google Cloud, and Azure. These platforms deliver a large number of infrastructure services through the **Infrastructure as a Service** (IaaS) model, such as virtual machines, databases, queues, and so on. The creation of all these resources can also be orchestrated, on the Amazon platform, through the use of *AWS Cloud Formation* (AWS Cloud Formation). Devs and ops can share the task of using IaaS services.

The cloud

Cloud computing is a model of ubiquitous, convenient, on-demand network access to a shared pool of computing resources (e.g., networks, servers, disks, applications, and services) that can be quickly provisioned and released with minimal managerial effort or interaction with the service provider (MELL; GRANCE, 2012). In short, it is a way of abstracting access to computational resources.

Cloud computing services can be offered to customers that are internal or external to the organization managing the cloud platform. A cloud is considered public when the customers are external (this market is in the hands of an oligopoly of US companies, mainly Amazon, Microsoft, and Google); or it is considered private when customers are internal, a situation in which the organization can use environments based on middleware systems such as OpenStack.

The term "public" here may sound confusing, as we are not talking about a service provided or organized by the State (e.g., public education, public transport). In this context, think of a shopping center: a space open to the consuming public, but privately owned, with the rules of the space governed by its owner.

It is also possible to use a **Platform as a Service** (PaaS), such as Heroku. In the PaaS approach, the platform is responsible for deployment, and developers do not know the virtual infrastructure that exists under the hood.

Services in the *serverless* architecture also seek to abstract servers for developers, going beyond PaaS by eliminating any notion of some type of continuously running process. Other differences of the *serverless* model are the transparency of the auto scaling process and the charge for computing usage.

These models suggest that there is no need for specialized operations teams (CUKIER, 2013); or, at least, that fewer people are needed for this (ROBERTS, 2018).

AUTO SCALING

Automatic scaling, or auto scaling, consists of the automatic increase of infrastructure resources (e.g., number of service instances) depending on the increase in the load on the system (which can be measured, for example, by CPU or RAM consumption). Example: a service starts with 10 instances available, but if the memory consumption of each instance reaches, on average, 80% of the available limit, then the service automatically starts 4 more instances to better support this increase in load.

When using IaaS, the provisioning of the environment is followed by the execution of scripts that install the application in the target environment. A deployment script can be written using Shell Script. However, configuration management tools like Chef and Puppet offer advantages by promoting operating system portability and idempotence mechanisms, which are difficult to achieve with Shell Script. An example of a Chef construct that leads to portability is the use of the *package* resource, which resolves to a concrete package manager, like Apt, only at runtime (the package manager is the

software that takes care of the installation of programs in the operating system). An example of Puppet's idempotence mechanism in the *service* resource is declaring the desired end state of the service as *running* rather than writing a command to start the service.

> **IDEMPOTENCE**
>
> An idempotent action is one that, when repeated, does not change its effect. An example of an idempotent action is pressing the elevator button: you can press the button for the 4th floor seven times, and it is the 4th floor that we will reach (the desired final state). A non-idempotent action is to increase the TV volume with the remote control: if we accidentally press the volume increase button twice, the TV volume will have a different final state than the desired one.

Another alternative for deployment is containerization, implemented mainly by Docker. Docker containers resemble virtual machines. The main difference is that they are lighter, as they share the host system's kernel. Docker and related tools such as Docker Compose, Kubernetes, and Rancher specify a collection of containers and their interdependencies. These specifications generate container images in the build stage, which are later instantiated in the target environment. Docker is usually used more for deploying applications, but there are already attempts to use it even for deploying the infrastructure itself, as experienced with OpenStack (KANG *et al.*, 2016).

Containerization could be seen as complementary to the deployment script strategy (done by Chef or Puppet). However, in practice, these strategies appear to compete. A Chef script runs continuously on the target node and its success depends on the previous state of that target node. However, in containerization the entire containerized environment is generated in the build, so the environment is

destroyed and rebuilt with each new version of the software. When compared to the Chef strategy, for example, Docker produces faster and more reliable deployment, but at the expense of bigger builds.

Using configuration management (Chef or Puppet) in a complementary way to containers involves configuring the Docker environment and updating operating system programs, such as SSH. Another use is to manage configurations that vary in different environments (test, approval, production). Such a configuration cannot be embedded in container images, as the same image must be deployed to all environments. Among practitioners, although it is possible to find support for a complementary use of both, we understand that there is a tendency to favor containers.

However, here is an example in which the use of containers is not favorable: when the system to be implemented is an ecosystem of other integrated subsystems and these already existing subsystems may not have been created to be run in containers. A case that fits into this situation is the *Portal do Software Público* (The Brazilian Public Software Portal), which integrates systems such as Colab, GitLab, Noosfero, and GNU Mailman (SIQUEIRA *et al.*, 2018).

Still on provisioning, there are two approaches: the use of "lightly cooked images" versus "heavily cooked images". In the lightly cooked approach, a virtual machine image is instantiated, and Chef is run to install the application; whereas, in the heavily approach, the image already contains the entire application. The heavily approach is similar to the container strategy but uses virtual machine technology. Deploying using lightly cooked images is less reliable because it involves more external resources during deployment, which tends to increase errors and delays.

Another example of an infrastructure provisioning tool is Terraform. It provides a declarative approach by which the user specifies the desired state of the infrastructure (i.e., the infrastructure assets that must exist). When the configuration is applied, Terraform applies a plan that takes the infrastructure from its current state to the

specified state. Terraform has an extensible architecture with several *providers*, through which it is possible to provision resources in different infrastructure providers (AWS or Google Cloud, for example). Thus, a company can create customized providers that operate on local infrastructure (not in the cloud).

As the application evolves, the database structure also evolves. Traditionally, database administrators (DBAs) maintain the database structure. The adoption of emergent design (recognizing that it is not possible to anticipate the database structure at the beginning of the project) and the need for frequent deliveries have encouraged the use of database migration tools like Flyway. These tools control the automated execution of updates to the database schema in different environments, allowing developers to manage the database structure on their own. In this context, it is necessary to rethink the traditional role of DBAs.

Devs and ops share these tools for deployment. The main risk here is the lack of clarity of responsibilities, which can cause friction in organizations during DevOps adoption. Developers believe they do not control their application and feel like they are doing someone else's job. On the other hand, infrastructure personnel may judge that developers are incapable of performing some tasks or using some tools.

4.6 Monitoring tools

Monitoring tools generally allow monitoring of the application's non-functional properties, such as performance, availability, capacity, scalability, resilience, and reliability. Self-repair, alerts, log management, and monitoring business metrics are examples of tasks performed by monitoring tools; they relate to concepts in the *runtime* category of DevOps concepts.

Examples of monitoring and alerting tools are Nagios, Zabbix, and Prometheus. Examples of log management tools are Graylog and Logstash. Cloud services also play an essential role in ensuring non-functional properties of applications, as they provide elastic resources that can be allocated on demand. Additionally, it is common for cloud services to provide monitoring and alert services.

There is also a need to analyze the distributed trace of microservices logs. In other words, when an error occurs or there is some slowness in the system, it is necessary to understand the sequence of invocations between the microservices that caused the problem, the time spent on each service, as well verifying the error log of each microservice. Examples of tools that display this distributed trace are Jaegar and OpenZipkin.

The tendency to form cross-functional teams, combined with the expectation that developers should be responsible for the product, forces the use of these tools by the development team. So, once again, the responsibility for using and mastering certain tools is unclear.

Tip for choosing tools

As seen, there are many DevOps tools to choose from. When considering a set of candidate tools for adoption, the tip we give is to first understand well which ones are competitors and which ones are complementary. The categories of tools that we provided can help: tools aimed at supporting different actors, objectives, and concepts are complementary. Are you unsure whether two tools are competitors or complementary? Initially consider them as competitors, which will reduce the number of tools used. Finally, if two or more tools are competitors, choose one of them, but without worrying too much about which one is the best — in general, it won't make so much difference.

Actors involved in the use of DevOps tools

The variety of DevOps tools seems to challenge the idea of a single person with the "DevOps engineer" title. Even an entire cross-functional team may have difficulty to know all of this. Meanwhile, companies preserving the operations department may have difficulty in precisely defining which roles should use which tools. To complicate even more, new roles are emerging, such as the data engineer, who is responsible for maintaining the infrastructure on which big data and machine learning systems operate.

In our assignment of actors to tool categories, we saw that defining this association was not trivial. This is because some tools seem to encompass both the dev and the ops worlds. Furthermore, the meaning of ops in each case may vary, being about operational duties or infrastructure design. In other words, we understand that adopting certain innovative tools from the DevOps world contributed to the blurring of the division of responsibilities between areas.

The discussion about the division of operational tasks between different areas of the company will be the topic of Part 2 of this book.

Part 2: The different ways to organize DevOps

A difficult question to answer is how – and perhaps *if* – an organization should be restructured to adopt DevOps. In Leonardo Leite's doctorate, the authors of this book conducted empirical research (that is, observing reality) to identify the different ways of doing DevOps actually in use in industry, in addition to seeking a deeper understanding of these options. We report the results of our research in this second part of the book.

Capítulo 5
Overview

This chapter presents a high-level view of the different ways of doing DevOps.

Our research was carried out in the context of software-producing companies responsible for deploying the software they produce. For brevity, in this book, we refer to them simply as "organizations" or "companies." In essence, the methodological strategy of our research was to interview a considerable number of professionals working in the aforementioned organizations. We then interviewed 75 professionals working in 59 different companies. Furthermore, it is worth highlighting that, in our sample, we sought to strive for diversity: we interviewed workers of different roles, genders, and levels of experience working in companies from different countries, domains, and sizes (small, medium, and large). Due to a research strategy, the interviews were anonymous, so we cannot reveal the names of the companies.

In addition to Brazil, some of the countries where the interviewees worked are the United States, Germany, Portugal, Spain, France, Canada, and Italy. There were also interviewees working remotely

with international teams. A company's *domain* concerns the business supported by the software it produces. The domains covered by the analyzed companies include education, healthcare, logistics, telecommunications, public administration, development support, contracting, tourism, manufacturing, finance, mobility, gaming, defense, networks, semiconductors, Internet of Things (IoT), cloud computing, justice, real estate, big data, research, insurance, e-commerce, fashion, international relations, inventory management, vehicle automation, and team management. These companies include some big techs (the largest technology companies) and some unicorns (startups valued at more than 1 billion dollars).

The result of our research is a **taxonomy** (i.e., a classification) of DevOps structures, which concerns the different ways of organizing the division of labor of operational activities and interactions in relation to development and infrastructure teams (i.e., the different ways of doing DevOps). We say that this taxonomy developed in our research is a **theory** in the sense that it is an explanation of the world built with the procedures of a scientific method — in this case, using reports obtained from the real world. Note that this use of the word *theory* differs from its everyday meaning, sometimes used in the sense of "hunch", which would be better expressed by the word "hypothesis".

We present, at a high level, our taxonomy in the following figure. The main elements of the taxonomy are the DevOps structures. Furthermore, our taxonomy also includes **options** available for each DevOps structure. These options detail how the structures work, but it is possible that organizations adopting a certain structure do not present their corresponding options.

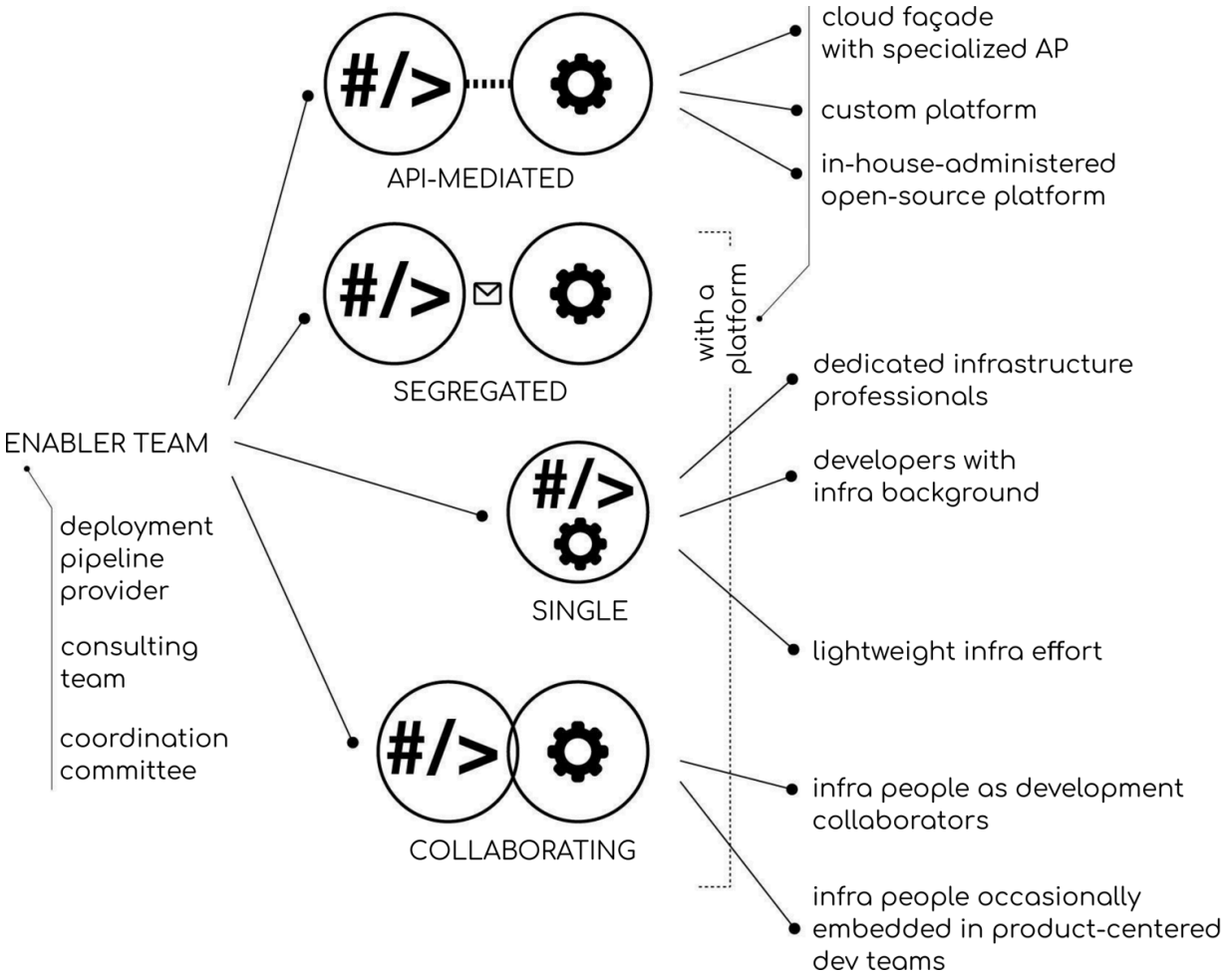


Figura 5.1: High-level view of our taxonomy of ways to organize DevOps.

Briefly, the DevOps structures present in our taxonomy are:

- **Segregated departments**, with highly bureaucratized interaction between development and infrastructure groups;
- **Collaborating departments**, focusing on communication and collaboration between development and infrastructure groups;
- **Single departments**, in which a cross-functional team (sometimes also called a product team) takes responsibility for software development and infrastructure management;
- **API-mediated departments**, where the infrastructure team (the platform team) provides highly automated infrastructure services to assist developers.

It should be noted that, in some cases, there is a process of gradual adoption of new structural patterns while old ones are abandoned. We call this situation a **transition** from one structure to another. In this situation, in the words of one interviewee, *"the new world still coexists with the old one"* and dissident forces of change usually act.

The next chapters present in depth each DevOps structure revealed in our interviews along with its options and an explanatory dimension consisting of its conditions, causes, reasons to avoid, consequences, and contingencies (strategies to deal with its disadvantages). We also present considerations about the **delivery performance** obtained by organizations using these structures.

> **Delivery performance**
>
> Delivery performance is a measure of an organization's ability to continuously deliver software, which in turn should be the goal of a DevOps approach. The delivery performance combines three metrics: deployment frequency, time from commit to production, and the mean time to recovery (FORSGREN *et al.*, 2018). This construct correlates with the ability of organizations to achieve both commercial goals (profitability, productivity, and market share) and non-commercial goals (effectiveness, efficiency, and customer satisfaction).

We also highlight that what we present in the next chapters is supported by the data obtained in our interviews, regardless of what the existing literature had to say. Furthermore, the next chapters also present excerpts taken directly from the interviews, so that you can have a bit of the feeling of being in the field talking to these professionals. These direct quotes from interviews are formatted with italicized text within quotation marks.

Capítulo 6
Segregated departments

In this structure, development and infrastructure departments are segregated from each other, with little direct communication between them.

The basic problem with this structure, vividly portrayed in the novel *The Phoenix Project* (KIM *et al.*, 2018) and witnessed by our interviewees, is the friction between these segregated departments, also called *silos*. Such friction happens because developers wish to constantly release new versions, while the operations team, aiming for stability, prevents these releases.

The DevOps movement was born in 2008 (DEBOIS, 2008) to deal with problems of this type. Let's now look at the characteristics that define this structure. Below, we present some consequences that we identified in our research interviews.

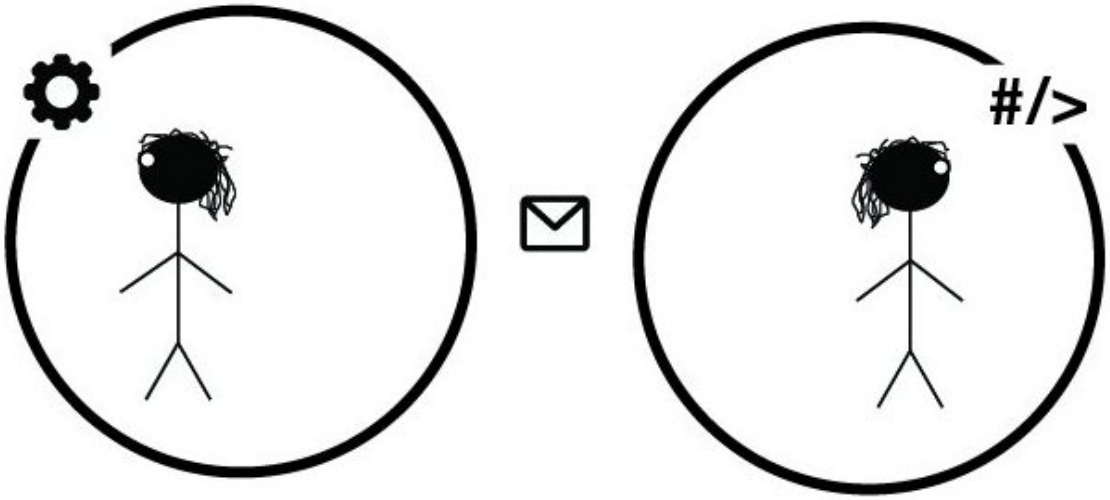


Figura 6.1: With segregated departments, developers and infrastructure people are each one in their bubbles. They don't interact much directly, and communication flows slowly through bureaucratic means (ticket systems).

The conditions, causes, reasons to avoid, consequences, and contingencies presented for each structure will be labeled A1, A2, and A3... where "**A**" means **assertion**. These statements are summaries of the findings of our research. They emerged from our interviews so that not necessarily all types of statements were identified for each structure. In the case of segregated departments, for example, we only have statements of the *consequence* type.

6.1 Characteristics of segregated departments

Developers and operators have **well-defined and differentiated roles**; as said by an interviewee: *"The limit was very clear: after sending the code, our work* `[as developers]` *was done"*. Therefore, there are no conflicts involving assignments. Well-defined roles and well-established workflows can reduce the need for direct collaboration between departments.

Each department is guided by its interests, seeking local optimization rather than global optimization. This problematic pattern was exposed by Goldratt and Cox (2014) more than 30 years ago. One participant told us about conflicts involving several departments: *"There's a big war out there... the security, governance, and audit groups still need to be convinced that the* `[Docker/Kubernetes]` *tool is good. The whole world uses it... but they don't understand... I get sick of it*".

Developers often neglect non-functional requirements (NFRs), especially security. In one company analyzed, conflicts between developers and the security group arise from disagreements over technical decisions. In other cases, developers have little contact with the security group and, consequently, *"are not aware of what happens there*".

Limited DevOps initiatives, focused exclusively on finding the best tools, do not improve communication and collaboration between teams or disseminate knowledge about automated testing. In one analyzed company, a "DevOps team" maintains the deployment pipeline and effectively functions as another silo. This sometimes culminates in delivery bottlenecks, thus fulfilling Humble's (2012) prediction about DevOps teams becoming a third silo.

Additionally, we've seen cases where developers have "infiltrated" people in the infrastructure team to skip steps in the corporate workflow. As one interviewee mentioned: *"When this group was formed to work with DevOps, the easy part was the outside guy* `[external consultant]` *, who knew Azure* `[the Microsoft cloud]` *and all the tools. This was not the problem, the problem was cultural because all hierarchies, powers, and needs were maintained."* Another interviewee claimed that: *"It's a question of power... I have power, I am important, I can deny you something... then you need to have a network, become friends with people, friend with the* `[ruling` *] party... for friends, everything; for enemies, death"*.

A **lack of appropriate test automation** has been observed in several segregated organizations. In an enterprise, developers only automate unit tests. Another organization was leaving test automation only to QA (quality assurance) people, which does not favor the practice of TDD (test-driven development). Although the structure of segregated departments is not the only one with a lack of test automation, developers in such a structure often underestimate the value of this practice and may even label a colleague as "incompetent" because he is *"taking too much time with testing "*.

About transitions

In our observations, no organization was transitioning to a segregated department structure, while most transitions observed were from segregated departments. That is, we can consider that there is a tendency for companies to abandon this structure.

6.2 Consequences of segregated departments

A1 - Devs do not have autonomy and depend on ops

In segregated departments, developers have very little autonomy, depending on the operations department to perform deployments and configuration changes, to diagnose problems in production, and even to find out about the existence of these problems in production. In the words of one interviewee: *"There was no autonomy for development and there was no autonomy for production"*. A dramatic example in this company was the need to open a ticket, to be answered by an operations person, so that developers could observe the application logs. Considering the cycle of "observing, analyzing, refining instrumentation, observing again...", a consequence of this was that developers sometimes took days to diagnose a problem in production.

A2 - Poor delivery performance

Organizations with segregated departments are less likely to achieve high delivery performance. As one interviewee said: *"Before the platform, in silos (...), deployment took a long time (approximately 3 weeks)"*. One of the reasons for this is bureaucratic approvals to deploy applications and evolve the database schema. This finding resonates with the thoughts of an interviewee: *"Will developers have an Agile mindset? If not, there's no point."*

On the other hand, we observed cases where poor delivery performance was not problematic. In one of the companies, applications are short-lived research experiments that do not need frequent updates. On the other hand, regular content releases (new phases of a game) usually do not require code changes, as *(i) designers* produce content with well-established tools and *(ii) correction patches* are rarely necessary thanks to comprehensive testing (even if manual). Network isolation policies, implemented for security reasons (for example: segregation between the repository

network, where the delivery pipeline lives, and the production environment), can also inhibit frequent deployments.

An important practical implication of this discussion is that if, for whatever reason, your organization needs to strongly segregate development and infrastructure professionals, then in that case the company should not place high delivery performance as a corporate goal.

A3 - Friction and blaming between devs and infra staff

Sometimes, the clear division of responsibilities in segregated departments ends up generating a certain *"accommodation"*; according to one interviewee: *"This clear division was abused to blaming games"*. In other words, sometimes individuals do not go as far as they could to solve a problem; they end up pushing the responsibility to another area so that the problem ends up taking much longer to be resolved. Or, if the problem is not resolved, *"accusations between devs and ops"* arise: each side is more interested in exempting themselves from possible blame than in solving the problem.

KEY CHARACTERISTICS OF SEGREGATED DEPARTMENTS

- Limited collaboration between departments;
- Culture of blame;
- Barriers to continuous delivery;
- Lack of automation.

Capítulo 7
Collaborating departments

The structure of collaborating departments focuses on collaboration between developers and the infrastructure team. This collaboration does not imply the absence of conflicts between these groups, but *"what matters is the quality of the conflicts and how to deal with them"*. We also call this structure "classical DevOps" because we understand that a collaborative culture is the main aspect of DevOps (according to our definition, "*DevOps is a collaborative, multidisciplinary effort within an organization...*").

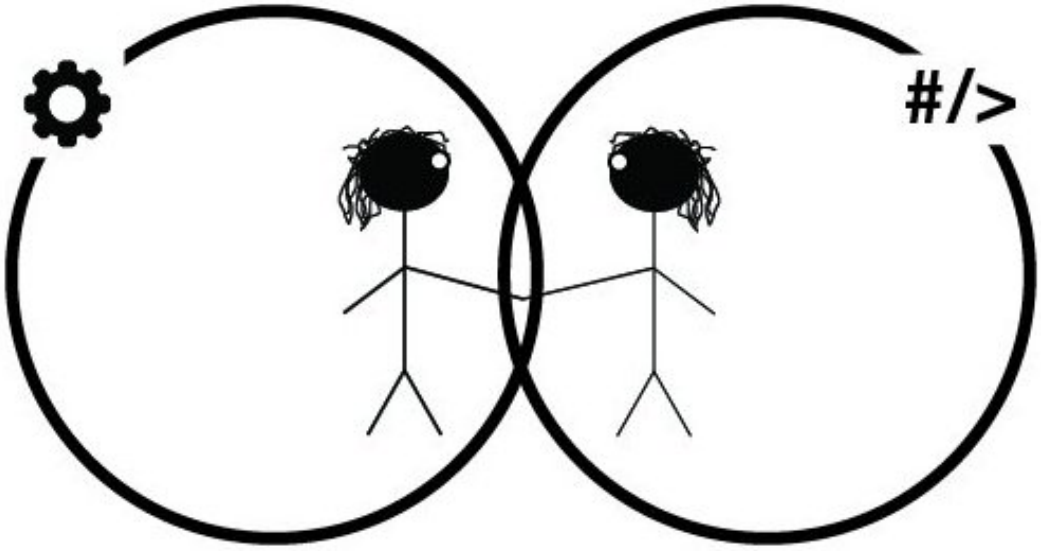


Figura 7.1: With collaborating departments, development and infrastructure professionals in different departments seek to work together, even if it is not easy, through direct contact and close collaboration.

7.1 Characteristics of collaborating

departments

We have observed that, in collaborating departments structures, many practices promote a culture of collaboration. We saw the sharing of database management: the infrastructure team creates and optimizes the database, while developers write SQL queries and manage the database schema. We have heard, in some cases, about collaboration between the development and infrastructure teams even before the product is deployed. One participant mentioned that the developers most concerned with architectural issues or non-functional requirements, the "guardians", stay in close contact with the infrastructure team. Another participant highlighted: *"Development and infrastructure teams participate in the same chat; it even looks like they are all part of the same team"*. In this structure, developers also support the product in its initial production phase. In other words, in summary, in collaborative department structures, the development and infrastructure teams are **partners who are helping each other**.

Developers feel relieved when they can count on the infrastructure team, since *" in the product team you kind of don't have time to worry about so many things"*. One participant claims that his previous job in an excellent company based on cross-functional teams was a much more stressful environment compared to his job at the time of the interview (another recognized excellent company, but adopting collaborating departments). On the other hand, **the stress level can be high for the infrastructure team**, especially *"if the application is poorly designed and performs poorly"*. In other words, *"operations people are the first to get f..."*.

In this structure, the success of the project depends on the **alignment of different departments**, which is not simple to achieve. In a company, different teams understand the organization's objectives and the consequences of possible problems (such as incorrectly computing amounts in the order of millions of

dollars). Another interviewee argued that alignment happens when employees focus on problem solving rather than definitions of roles.

Development and infrastructure teams share responsibilities for non-functional requirements. For example, in one analyzed company, both teams were very concerned about low latency, a fundamental requirement for the product.

Typically, **the infrastructure team is the front line in monitoring and resolving incidents**. However, if necessary, developers can be summoned to collaborate on the resolution. In one of the examined companies, monitoring alerts are assigned to the infrastructure team, but a copy of each alert is sent to developers. In some cases, developers, unlike infrastructure staff, do not work after hours. In another company, *"developers are usually not called after hours because the development team leaders are sufficient"*.

Some observed organizations **rebranded their infrastructure teams as DevOps or SRE teams**. There is some debate about the difference between DevOps/SRE departments and traditional operations departments. However, in our observations, these DevOps/SRE teams are simply infrastructure teams that are more inclined to embrace automation and collaborate with developers when compared to infrastructure teams in the segregated structure.

This need to embrace automation puts pressure on some infrastructure professionals, who must learn new skills, especially in large organizations. One participant in our research, a consultant, saw it as problematic that *"many infrastructure professionals don't want to go into cloud computing"*.

Humble (2012) hopes that with a culture of collaboration between developers and the infrastructure team, **a DevOps team will not be needed**. For us, this criticism refers to DevOps teams with dedicated people, as we have seen in a company, as they behave like a new silo. However, at another company, we found a functional

DevOps team acting as a strategic decision committee — a forum for leaders from different departments. We also found DevOps groups working as guilds (KNIBERG, 2014), spreading knowledge among different departments.

7.2 Options for departments that collaborate

Infra people as development collaborators. Infrastructure people contribute to application code to optimize system performance, reliability, stability, and availability. Although this approach requires advanced programming skills from infrastructure professionals, it can prove to be a successful strategy for maintaining large-scale systems, such as those seen in an analyzed organization.

Infra people occasionally embedded product-centered dev teams. Infra professionals join development teams for a limited time, mainly at the beginning of the product's life cycle, in accordance with the development teams' project schedules. However, this integration occurs informally, without changing hierarchical structures. In one company where we saw this pattern, this integration took about six months. And even in this period, the infrastructure person may act only part-time in the dev team.

7.3 Conditions for collaborating departments

A4 - Sufficient number of infrastructure people to align with dev teams

The structure of collaborating departments is centered on promoting a culture of collaboration. A recommendation from the literature for promoting such collaboration is to invite infrastructure people to

agile ceremonies (daily meetings, reviews and retrospectives), so that infrastructure people understand the context of the project and feel part of the delivery.

Even so, organizations should be aware that such an initiative will likely fail if the company has many more development people than infrastructure people. In the beginning, infra personnel will begin participating in agile ceremonies, but their task queues may not decrease. On the contrary, ideas arising from these meetings can even increase the workload for them. After some time, these infra staff may find it better to stop attending meetings and things will work as before. This is what happened in one of the analyzed companies: *"We tried this `[collaborating departments]` approach for a while, and it didn't work. (...) The infrastructure team was small and had to be in several teams at the same time."* Another interviewee said: *"We had more engineers working in development teams compared to the infra team (...) So they `[devs]` were stuck, while we `[infra]` were handling incidents, and we didn't have time to help `[devs]` people."* In contrast, in a successful case, *"there are enough SRE teams `[i.e., infra people]` to align with development teams"*.

Suppose your company has a low ratio of infrastructure people to development people. In this case, one implication of our findings is that it may be better not to adopt a strategy of collaborating departments. If this path is insisted upon, directors must be aware that some additional measures are necessary, such as giving more autonomy to developers (A14) so that infrastructure staff are not overloaded.

A5 - Support from senior management

Unlike other organizational transformations, professionals from different departments can voluntarily initiate closer collaboration. Although this is a valid start, some interviewees warn that *"part of the success of a DevOps/SRE team depends on the support of senior management. It only works if you have sponsorship from management or leadership"*. In other words, without corporate

sponsorship, many of the cultural transformations achieved voluntarily may be lost as soon as pressure for immediate results arises in a way that inhibits the communication and coordination necessary for collaborative action.

7.4 Causes of departments that collaborate

A6 - It is easier to be collaborative in a small or medium-sized company

Ticket systems in large companies produce relations between teams with a certain level of impersonality, so it is difficult to be collaborative in this context. Sometimes, who opens the ticket does not even know who is answering it. It is not a condition, but small and medium-sized companies end up providing a much more favorable environment for direct collaboration between professionals. One interviewee said: *"Our company has been a small company for a long time. (...) I could open a document that the CTO was working on and comment* `[on the document]`*, it was very easy to get involved in whatever aspect you wanted to be involved in."* Another interviewee, from the infra area, said that the fact that there were few developers in the company was a motivating factor for him to expand collaboration.

A7 - Trying to avoid the delivery bottleneck

In many companies with segregated departments, infrastructure personnel end up being a bottleneck in the delivery path: a deployment request or a configuration change can take days to be answered. In this way, many people understand collaboration as a mechanism for the infra area to share responsibilities with development, thus relieving infrastructure professionals (*"decentralize to grow"*). Summing up the issue, one interviewee

motivated collaborating departments as a *"partnership trying not to delay the creation of infrastructure"* so much.

A8 - Bottom-up initiative with later top-management support

One of the interviewees in a company with collaborating departments reported: *"It started more with the technical staff, and then the senior management embraced the cause, then it kind of flowed, it escalated to the senior management staff"*. Unlike other organizational transformations, professionals from different departments can voluntarily initiate closer collaboration. It is much more difficult, for example, to voluntarily start building a platform, as it requires a large investment. The formation of single departments implies changes in hierarchies that must be approved by the board in advance. However, as already explained (A5), the chances of success for collaborating departments without subsequent support from senior management are low. The involvement of directors and CTOs is so important that some interviewees even explicitly mentioned the names of these leaders who supported the DevOps transformation in their companies.

7.5 Consequences of collaborating departments

A9 - Growing interaction inter-areas

Obviously, an expected consequence of collaborating departments is an increase in interaction between areas. However, more communication is not necessarily better communication. Therefore, care must be taken to ensure that this interaction does not occur in the form of an obligation to go to meetings just for the sake of the obligation itself. One interviewee commented that, although this

interaction between areas is not always desirable (*"it would waste a lot of time in coordination"*), it is very useful in the initial phase of a project: *"This strategy is meeting intensive, you have a lot of meetings with the team, so it's very good for discovery, but it takes a lot of time"*. Another interviewee, although defending the API-mediated model, recognized that some mix between the models could be desirable, since this would *"improve both development and infrastructure, so that they share knowledge and learn from each other "*.

A10 - Precarious collaboration with overloaded ops

As already explained (A4), the participation of infra people in agile development ceremonies has pros and cons. The disadvantage is the risk of taking too much time from infrastructure professionals and leaving them even more overloaded. One interviewee, a development manager, commented: *"So sometimes you are served super quickly because there is little demand; sometimes everyone decides to do a certain thing, so the person becomes extremely overwhelmed. No matter how much they collaborate, speed has a limit. It depends on the person's productive capacity"*.

To address this problem, the size of the infrastructure area must grow as the development teams grow (A4). However, this is not necessarily true for other strategies. With API-mediated departments, there is no need to scale the infrastructure team depending on the number of developers since, in this case, the infrastructure staff is not burdened with daily interaction with the development staff.

A11 - Discomfort with blurred responsibilities

The structure of collaborating departments is centered on promoting a culture of collaboration. However, sometimes problems arise due to unclear responsibilities for some tasks. We have seen cases where one group had expectations about what another group would do and, in the end, those expectations were not met. In the view of an

interviewee who defends single departments: *"The team is split into two, a dev team and an infra team. Problem: if there is an issue, it may not be clear who owns the issue – and this generates inefficiency. Such a division harms ownership"*.

Companies that want to foster a culture of collaboration must consider addressing this issue. Perhaps imposing strict rules about responsibilities can make collaboration difficult. Thus, a possible solution is to align different groups with a common goal, so that people focus more on "*what can I do to achieve the goal?*" rather than on "*what is my responsibility?*". In one company, this alignment arose from the awareness of professionals from different areas that failure to fulfill a mission could cause major financial losses for the company. Even so, caution is necessary. Sometimes people feel it is unfair when some goal is not achieved but their groups "did their part", especially when achieving the goal is linked to some reward.

A12 - Waiting: infrastructure is still a bottleneck

Collaboration between areas does not seem to be enough to achieve high delivery performance (*"so you may end up having to wait a while to have some interaction with the area you need"*). Many analyzed organizations with collaborating departments did not have high delivery performance, even some of them possessing automated delivery pipelines. One possible reason for this is the lack of adequate test automation. Achieving such an automation is a difficult goal as almost all developers must master this discipline.

Another limitation to high delivery performance is the adoption of release windows. They exist to reduce deployment risk by restricting software delivery to periodic time intervals, reducing the negative impact of any deployment problem. Release windows are adopted due to the massive number of users or the financial criticality of the systems. They can also be the result of fragile architecture or a monolithic architectural style, given that, in this case, a deployment has a greater risk of affecting the system as a whole. Even so, there is no consensus on these windows and it is a controversial topic:

while some developers do not support release windows (as they make deliveries difficult), others consider them necessary.

A13 - Automation supports collaboration

In our interviews, we realized that an effective collaboration point is precisely the collaboration to build automation: increasing communication between areas and keeping the same old ways of working is not promising. One interviewee said how automation initiatives helped to bring the areas closer: *"And they started to implement continuous integration pipelines, bringing together, integrating the areas of development, operations and infrastructure*".

The literature corroborates this point. The Brazilian Public Software Portal project was an example of how the construction of an automated delivery pipeline helped to bring teams from different organizations closer together (SIQUEIRA *et al.*, 2018). In contrast, other authors (NYBOM *et al.*, 2016) reported a scenario of departments trying to collaborate, but with a lot of friction and frustration (A11). In particular, investment in automation in this scenario had been meager.

7.6 Contingency for collaborating departments

A14 - More autonomy to devs

We have already talked about how companies generally have a reduced number of infra staff and how the initiative of collaborating departments can further overload these infra staff (A10). Obviously, this scenario is undesirable, as the infrastructure would continue to be a bottleneck on the delivery path (A12). An important strategy to deal with this problem is to give more autonomy to developers.

In some cases, it may be that the infra staff do not have full confidence in the development people to work directly in production. One of the interviewees said that there was an attempt to grant access to production to developers, but that this access was revoked after some problems, including developers losing access keys to these environments. However, in these cases, more autonomy in non-production environments will reduce the burden on infrastructure personnel. The same interviewee also told us: *"All day long people were asking 'press this button for me'. So now you have one person in each squad who can press that button; at least in most environments, except in production"*. This interviewee also talked about how developers had the autonomy to create deployment scripts, which were freely applied in non-production environments. These scripts were subsequently screened by the infrastructure staff, who were still responsible for executing them in production. This case also exemplifies development and infrastructure areas collaborating on deployment automation (A13).

KEY CHARACTERISTICS OF COLLABORATING DEPARTMENTS

- Intense collaboration between developers and the infrastructure team;
- Seeking for alignment between departments;
- Blurred responsibilities.

Capítulo 8
Single departments

In the context of this book, we define a cross-functional team as one that assumes responsibility for both software development and infrastructure management. This structure aligns with Amazon's motto "*You build it, you run it*" and the *squads* at Spotify. This scheme gives more freedom to the team, along with great responsibility. As one interviewee described: *"It's as if each team was a small company, having the freedom to manage its budget and infrastructure"*. This unified team is also called the product team.

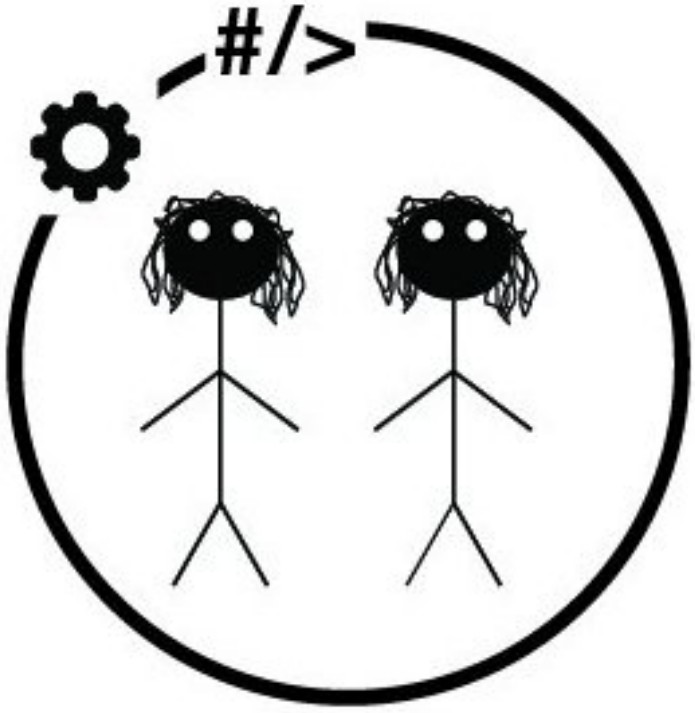


Figura 8.1: A single department takes care of development and infrastructure.

We call this structure single departments because, in addition to the case of the cross-functional team, we also apply this structure to the

situation in which each department of the organization, containing multiple development teams, has a dedicated pool of infrastructure professionals. In other words, the infrastructure professional may not even be on the same team as the developers, but infrastructure and development professionals have at least the second hierarchical level of leadership in common, so they are not that far apart. We are talking about departments unifying development and infrastructure people, but not necessarily about a single department in the company.

8.1 Characteristics of single departments

Independence between teams can lead to misalignment. Lack of communication and standardization between single departments within a single organization can lead to duplicated efforts: multiple teams producing similar things. However, this is not always a problem. In a very large company that we analyzed, diversity of solutions across different teams was seen as a way to boost innovation. In any case, to maintain alignment, a strategy seen in another company was to make developers know the business in depth.

It is difficult to ensure that a team has all the necessary skills. For example, we interviewed cross-functional teams without any infrastructure experts. One participant acknowledged that *"there is a lack of knowledge"* in infrastructure, deployment automation, and monitoring. One possible reason for such adversity is that, as one interviewee revealed to us, it is difficult to hire infrastructure experts and senior developers.

We expected that in cross-functional teams, infra specialists would have a lot of idle time compared to centralized pools. However, we found **no evidence of expert idleness**. In an interview, we heard that, quite the opposite, infrastructure specialists were too busy to

be shared with other teams. But to avoid such idleness, in one case we saw infrastructure specialists programming new features in their "spare time".

The majority of cross-functional teams interviewed worked in small organizations. After all, there is no point in creating several teams in embryonic companies.

8.2 Single departments options

Dedicated infra professionals. The department or team has specialized personnel for infrastructure tasks. In one analyzed organization, one employee was a specialist on physical infrastructure, while another one was "the DevOps" (taking care of the deployment and monitoring pipeline). In this scenario, infrastructure specialists become the front line in incident resolution and monitoring.

Developers with infra background and attributions. The team includes developers with knowledge in infrastructure management. These professionals are also called full-stack engineers or even DevOps engineers. One participant of our research was a full-stack engineer and claimed to know *"all the technologies involved: front-end, back-end, and infrastructure; so I'm the person capable of connecting all of them and dealing with problems when necessary"* . Another participant, a consultant, was skeptical about full-stack engineers and declared that *"these people are not capable enough"*. He complained that developers often don't know how to tune application configuration, such as database connections. In fact, we heard about problems regarding database connections at another company without infra specialists.

Lightweight infra effort. Sometimes, regardless of the development effort and operational overhead involved, software may

need very little effort to prepare the infrastructure. In this case, you end up not needing infrastructure specialists because of the little infrastructure effort required. The places where we saw this case were startups that had just launched their applications, being more focused on validating the business value of the application than on building a scalable infrastructure.

8.3 Condition for single departments

A15 - Enough ops for each dev team

In the context of dedicated infra professionals, the biggest challenge is meeting the condition of getting infra people on each product team. In a company with API-mediated departments, the interviewee stated that the single-department approach would be unfeasible, as *"I don't have enough people to put a `[infra]` person on each team"*.

8.4 Causes of single departments

A16 - Startup scenario

In startups, the emergence of cross-functional teams is common and expected. As one interviewee said: *"So, we are a startup, so we have to do several different things in the same role, which would normally be several different roles"*. In other words, the few people available end up playing multiple roles, with development and infrastructure being a common case.

Furthermore, an interviewee explained to us that the essence of a startup encompasses risk: it is not known whether the product being

developed will succeed, therefore large investments in a scalable and high-performance infrastructure are not worth it (the priority is to validate the business hypothesis). As a result, it is also common for startups to use cloud services to manage infrastructure, as these are services that typically have low operational overhead and low initial costs (although down the road they may cost more, according to comments we have heard).

A17 - Cloud services decrease the need for ops and infrastructure staff

There is also a perception that cloud services, by reducing operational overhead, end up eliminating the need for operations teams or infrastructure specialists. One interviewee told us: *"We use Heroku and several PaaS* `[platform as a service]` *vendors. This decision was made early on, knowing we were short on people"*.

Of course, on the other hand, we also saw several companies using cloud infrastructure managed by experts (who even complained about the ineptitude of developers for this). With the increasing number of cloud services available (more than 200 on AWS alone), it may not be so credible to expect all this knowledge from developers.

A18 - Delivery velocity

This is certainly the greatest asset of a cross-functional team: the elimination of handoff. In other words, the development team does not need to wait for the infrastructure team to finish another task being done for another development team. This speeds up delivery velocity.

Considering this aspect, one interviewee was insightful in noting that this strategy does not necessarily need to apply to the entire company, but only to more critical projects (*"We cannot fail to remember that this is one of the few projects that brings resources to the company . So it's the flagship"*).

8.5 Reasons to avoid single departments

A19 - Unsuitable for enforcing corporate standards

With infrastructure staff spread across different departments, many managers worry about different teams taking different approaches to infrastructure management. Organizations considering moving to single departments should be aware of this concern.

Enforcing corporate standards may require more effort for single departments compared to the scenario with a centralized infrastructure group. Therefore, top management must evaluate: if the application of standards is really necessary, is it worth adopting single departments? If the organization imposes corporate standards on single departments, the company must carry out effective planning, considering the need for innovation. Is having a homogeneous infrastructure really necessary?

The main concern of managers with heterogeneous infrastructure is about increasing onboarding time when professionals move from one team to another. But would this concern really be valid or a priority? Do employees really switch teams that often? Plus, don't employees move frequently from company to company these days? If so, perhaps professionals should be prepared to adapt to new technologies anyway.

Below are some excerpts from our interviews on the subject. It is interesting to observe "control" as an objective of standardization.

- *"So there are these advantages in a little of standardization. I think super independent teams, isolated from each other, tend to get a little confused* `[about standardization]` *. Standardization of both infrastructure and programming language, so that things don't get out of control (...). This has certainly been a concern."*
- *"To speed up you need a certain standard. There's no point saying 'Ah, everyone does what they want, everyone puts it into*

production'. I find it difficult for a large institution, with a reputation to uphold, to work in this way. It won't be like that..."

- *"They are afraid of opening too much and we will start to lose control over the best practices and care with security, with availability, with attacks... These `[infra]` people are going to lose some of their roots and become less connected to their areas of expertise."*

A20 - More costs

There is a view that single departments cost more, and this is for several reasons. Paying a lot for specialized infrastructure professionals, potentially with some idleness, is one of them. Independent teams applying parallel efforts to solve similar problems is also seen as a type of waste that is a cost that could be optimized.

But it's not just competing development efforts that are the problem. In single departments, there is also the idea that each department has a completely independent infrastructure. An example: if the hype in the company is to use Kafka, each department will have to manage its own Kafka cluster. One interviewee saw this with concern: *"Adding a Kafka cluster to everything I have... It's too much for each one there, it's not cost-efficient*".

8.6 Consequence of single departments

A21 – No infra defaults

An interviewee from a company that has highly independent teams told us: *"There are standards within teams, which are different from the standards of other teams. But a standard among the different teams, that doesn't exist*". This worries some managers: such freedom can be used without the responsibility of considering the

cost of long-term maintenance. As an example, managers don't want *"people to use exotic databases... no one knows how to maintain them"*. This is related to the subsequent need to hire specialized labor, which can be expensive. Or worse: diverting the focus of professionals with knowledge in "exotic technology", which is also a form of cost by potentially delaying other projects.

8.7 Contingency for single departments

A22 - Improve infrastructure skills in-house

In the approach in which every single department has developers with infrastructure knowledge, a challenge is to motivate and promote this knowledge among at least some developers. In general, *"the solution is to train in-house"*, *or "training the folks in the long-term*", with one of the preferred practices for this being *tech talks*. In this case, it is recommended the company culture, including the selection process, to value the ability to learn how to learn.

KEY CHARACTERISTICS OF SINGLE DEPARTMENTS

- Autonomous teams or departments in charge of both infrastructure development and management;
- Requires professionals with different skills in the same team or department;
- Unsuitable for imposing unified standards across different teams or departments.

Capítulo 9
API-mediated departments

Platform teams provide highly automated infrastructure services that can be used autonomously by developers to deploy and operate their systems. The infrastructure team (the platform team) is no longer a "support team"; it behaves like a product team: the internal infrastructure platform is its product and developers are its (internal) customers. In this setup, infrastructure specialists need programming skills, while product teams must operate their services.

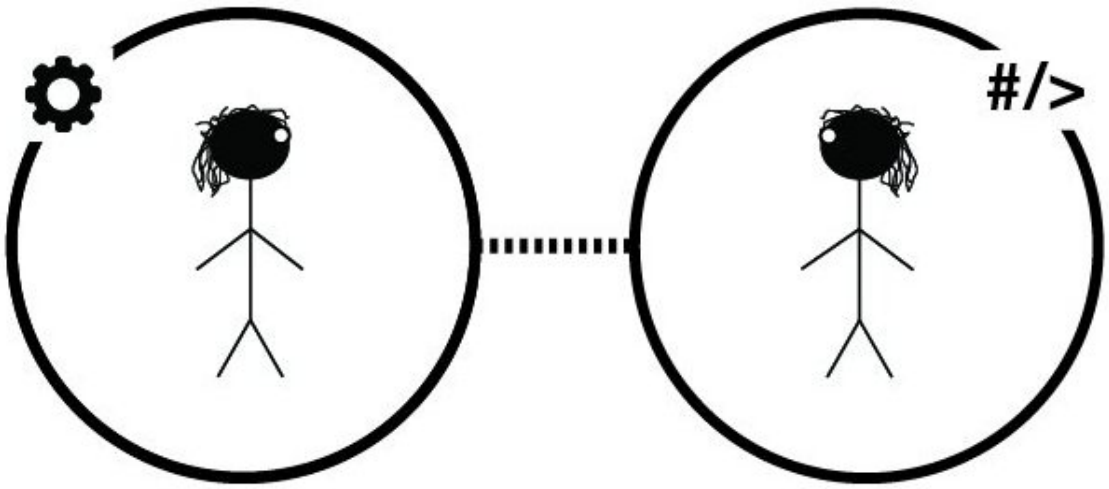


Figura 9.1: The platform team provides automated services to be used, with little effort, by developers through a well-defined interface (the platform API mediates the interaction between developers and the platform team). The platform team and developers are open to listening and supporting each other.

Direct collaboration between people in the development and infrastructure departments occurs sporadically, in certain necessary moments. In general, on a day-to-day basis, the mediation of development requests for the infrastructure is done in an automated way, which is why we call this structure *API-mediated departments*. In practice, this means that developers will be able, for example, to

deploy new versions of their services through commands that can be triggered automatically by the deployment pipeline.

> **APIs**
>
> An API (Application Programming Interface) is the description of the interaction possibilities that a system offers for other systems to activate it programmatically. In many contexts, the term API has already become synonymous with REST API (or RESTful API), which are APIs activated remotely (via the network) using the HTTP protocol. But in this book, we use API in a broader sense. Writing a structured file (YML, for example) can be considered an interface for the automated activation of a system (such as the deployment pipeline).

We can say that this is the most valuable chapter in this book, as the phenomenon of platform teams is relatively recent, after the emergence of DevOps, and not yet covered as much in the literature (we were the first ones to report about it in the peer-reviewed literature).

9.1 Characteristics of API-mediated departments

The infrastructure team is no longer required for operational tasks. Operational tasks are automated by the platform. Consequently, you can't simply call platform team members "operators" since they also design the infrastructure solution. We emphasize that, in other industries, "operator" is a title given to menial workers (see section *And who is the operator?* of the chapter *What is DevOps?*).

The platform **avoids the need for product teams to have infrastructure specialists**, as would be the case for cross-functional teams. One interviewee expressed a desire to understand how the platform works "under the hood", which indicates how well the platform relieves the team from mastering infrastructure issues (that is, she *would like* to understand more, but it was just a curiosity). On the other hand, as developers are responsible for deployment, they must have some basic knowledge about infrastructure and the platform itself, unlike what happens in a segregated structure.

Product teams are responsible for the non-functional requirements (NFR) of their services. They are the first to get involved when there is an incident, and the issue is only escalated to the infrastructure team if it is related to an underlying infrastructure service. This situation differs from the scenario of collaborating departments, where generally infrastructure personnel are the first to deal with any incident, summoning developers only if necessary. Although the product team becomes responsible for the NFRs of their services, this is not a significant burden that developers try to refuse.

The platform itself extensively addresses NFRs through automatic scaling, monitoring, and other mechanisms. As stated by one interviewee, *"The idea is to make life easier for development teams: they get these monitoring panels for free, effortlessly"*. Furthermore, we observed infrastructure people willingly helping developers with service availability, performance, and security, which also motivates developers to take on this responsibility.

We found a **positive relationship between the adoption of API-mediated departments and high delivery performance**. One explanation for this is that this structure decouples the infrastructure and product teams, avoiding the infrastructure team to become the bottleneck in the delivery path. As stated by one interviewee: *"Now developers have the autonomy to go from scratch to production without having to wait for anyone*". This structure also

contributes to service reliability (another factor of delivery performance) by automating issues related to NFRs and holding product teams accountable for these NFRs and incidents. Therefore, we affirm that having a platform team is a promising way to achieve high delivery performance.

About **transitions**: No organization we analyzed was moving away from API-mediated departments. Combining this with what we have already seen about segregated departments (most transitions from segregated departments and none to it), we have even more evidence about the problems of segregated departments and the promises of API-mediated departments.

9.2 API-mediated department options

Cloud facade with specialized API. The platform deploys applications in third-party public clouds, such as Amazon AWS, Google Cloud, or Microsoft Azure. Although these clouds facilitate deployments when compared to physical servers, they still offer dozens of services and a multitude of configurations. The platform then ends up standardizing the use of public cloud providers within the organization, so that developers do not need to understand many details about the cloud — that is, the platform *" improves the usability of the `[cloud]` infrastructure"*.

We also observed a small company with a "platform mindset" but without the resources to build its platform. The infrastructure team prepared infrastructure-as-code templates encapsulating best practices so that developers would only need to provide some data for deployment.

Custom platform. The platform is built on top of internal physical servers, hiding infrastructure complexities from developers, such as the use and even the existence of Kubernetes. It is in this option

that more components of the platform are developed by the company itself.

In-house administered open-source platform. Here, the platform is free software, which can be deployed on company servers or even on virtual machines in the public cloud. One analyzed organization used Rancher, a graphical interface for developers to interact with Kubernetes.

With a platform. The organization has a platform that can provide deployment automation, but it does not follow the human interaction and collaboration patterns envisioned for API-mediated departments. One participant in our research developed a *"autonomous IaaC `[a more automated infrastructure]` for integration and deployment with Google Cloud"*, which provides platform capabilities for other developers on the team. However, as in this case there was a single cross-functional team, it would not make sense to talk about a "platform team".

In one analyzed case, we considered an organization to have segregated departments, even with a platform team, as developers and the platform team had a relationship that was worn out due to a lack of trust. The other options we mentioned, for API-mediated departments, can also be used to classify organizations “with a platform”.

9.3 Conditions for API-mediated departments

A23 - For medium to large-sized companies

Developing a platform requires an upfront investment (A25). For small companies, it is usually unfeasible to dedicate a group of people to such an investment. On the other hand, for some people, medium and large companies should avoid the role of full-stack

engineers and promote the specialization of activities: *"When we have a large company or product, and you need to scale, and you start to have thousands of microservices and so on, it is very difficult to focus on developing the product and also developing the infrastructure. At some point you have to choose one of them, you have to specialize"*. This specialization may be unfeasible for very small companies as it requires more hiring, but that is what API-mediated departments provide.

A24 - Sponsorship from senior management

Even if minimally, the creation of a platform team affects the company's organizational chart (A28). You don't create a platform (which requires a lot of operational effort) voluntarily in the "free time" of teams already committed to other tasks. Furthermore, development teams need some corporate approval to deploy their services on this platform. For these and other reasons, the creation of a platform team must be sponsored by the company's senior management, even if as a pilot project.

Some interviewees told us how the support of senior management was fundamental: *"The CEO of the company, he is the great motivator here, he cheer up and engage the folks, he understands that it has to be this way, and makes the areas report to the centralized infrastructure area"* . However, when trying to sell the idea of a platform to management, you need to keep in mind what interests them: *"money"* (that is, the company's profit).

A25 - Upfront investment

When investing in the API-mediated departments approach, you need to be aware that building a platform requires *"a much larger initial investment; it's a medium-long term bet, you won't see gains that quickly"*. And investment doesn't just mean money, but also time from the involved people. In other words, it is necessary to protect platform engineers from everyday interruptions, so that building the platform can truly be a priority.

This high initial investment was a factor that discouraged one of the analyzed companies from continuing to build a platform. According to our interviewee: *"So we decided to go for the awareness and analysis side, instead of having a high investment, making a tool that will take a good few months to get ready and running"*.

A26 - Infra people with coding skills

Those who build the platform are people called platform engineers. Especially for custom platforms, developed entirely in-house, their skills must go beyond those typical of infrastructure professionals, as they need to develop the platform. But even when it's not a custom platform, a "developer mindset" will be imperative when dealing with various tools that follow the infrastructure-as-code paradigm. In other words, once infrastructure assets start to be expressed as code, professionals who maintain this infrastructure must be concerned with practices typically dominated by developers, such as the use of version control and the use of readable names in the code, for example. One strategy we observed to meet this need was to hire developers for the platform team.

As one of our research participants said: *"Everything is code. So it's not about being a infrastructure member or a developer. They are just programming different layers of the stack; but at the end of the day, they are programming. In the future everything will be code "*. With this understanding that *"in the future everything will be code"*, another interviewee, a platform team manager, went further, giving a harsh opinion: *"The guy who is a sysadmin and doesn't know the basics of development, debugging a code, it is doomed to cease to exist"*.

Since the platform must be developed, maintained, and operated, the discussions in this book also apply to the operation of the platform itself: would it be appropriate to delegate the operation of the platform itself to people focused on that mission? Or should the same engineering people who develop the platform operate it? This is an open point that we did not explore in our research.

9.4 Causes of API-mediated departments

A27 - Delivery bottleneck in infra management

Just like for collaborating departments (A7), the main objective here is also to make the delivery step no longer a bottleneck in the software production process. "*We want autonomous teams that can solve their problems independently. This avoids blockages in the development process, blockages that would hinder the time to market."*

A28 - Low impact on the organizational chart

Some professionals argue that their organizations should adopt single departments to eliminate handoffs and bottlenecks. However, in a large company with a centralized pool of infrastructure professionals, it is difficult to break the hierarchy and disperse infrastructure personnel across different departments. When carrying out such a transformation, many people will lose their positions as bosses, which can generate friction. A developer working in a context of API-mediated departments told us that *"cross-functional teams don't fit into the company structure, which has a development department and an operations department, devs and ops in totally different structures*".

HANDOFFS

Let's say a development team asks the infrastructure team to make a change to the configuration of an application. As long as the infra team does not respond to the call, the development team will be blocked, and unable to continue with that task. Changing the configuration is very quick, an infra person can do it in 5 minutes; however, this person has dozens of similar requests in front of them, so it will take some time, say three days, for the request to be fulfilled. These three days is the waiting time in the infra team's task queue. We call this time *handoff*, or *handover*, which in short is the transfer time of a task from one team to another.

On the other hand, setting up an experimental platform team can be cheaper and bring less resistance. The platform team can be formed as a group within operations, possibly also incorporating some developers. If there is no success, people will be able to more easily return to their original groups. If the platform achieves some initial success, the team can grow little by little, considering that adding more services or clients to the platform does not require adding proportionally more platform engineers (A38). Therefore, hierarchies are gradually affected.

A manager of a platform team revealed to us: *"You don't have managers willing to loose their key employees and you don't have the possibility of hiring people* `[public sector context]` *. The platform team was ideal because we were able to move people within the company, without generating a huge impact on other areas, to set up a platform. There are around 15 people on the platform, it wouldn't be enough to the cross-functional approach"*. Another interviewee also corroborated this view when talking about API-mediated departments: *"It's the best model for the size of the company. I don't have enough people to put an* `[infra]` *person on each team. I think this allows us to have a leaner platform team"*.

A29 - Fosters continuous delivery

Several participants of our research expressed conviction that the infrastructure platform model promotes continuous delivery (the ability to deploy at any desired time), agility, and rapid change. One of them said that moving to a platform *"enabled us to continue the accelerated growth that a startup needs"*. Another interviewee understood that the strategy was *"the best one to facilitate continuous delivery and agility, as far as possible, for infrastructure"*.

Another interviewee was explicit about associating the platform's API with the company's rapid growth and presented the following explanation for this success: *"It's as if each group focused on what they like to do most and what they do best. So, both groups can deliver what is the necessary"*.

A30 - A hero or visionary

Even if the creation of a platform is unfeasible without minimal support from senior management (A24), the initial development of the platform is often thanks to the effort and entrepreneurial capacity of a visionary leader in the company (*"There was a guy who was the biggest responsible for the platform"*), at least until the platform's value hypothesis is validated and it is clear to the interested parties. It is this "hero" who often bears the risk of the enterprise (*"You can do what you want, but if things go wrong, you will handle it"*, the board told to one of these heroes).

A31 - Emerged as the best solution

Although the viability of an infrastructure platform depends on the sponsorship of senior management (A24), initially it is common for this solution to emerge from below, proving to be better than other tried alternatives. One interviewee said that there were no major debates before the construction of the platform, concluding that *"it must have been an emerging behavior"*. Another was explicit about the platform not initially being a company decision: *"It was more*

about exploring new things, how to automate things". In another company, an alternative that received a good investment, but that did not prove to be as transformative as the platform, was automation within the infrastructure group, but still in the model in which developers continued to open tickets.

From these reports, we can learn a lesson: it is wise to request senior management support for the platform when there is already evidence that the solution is promising. However, these are cases of pioneers in structuring API-mediated departments. We hope that this book will help engineers advocate for a platform, when appropriate, without necessarily having to go through a long process of trial and error.

A32 - Multiple products, teams, and customers

Many companies, especially startups, are single-product companies. In these circumstances, when the product still has a monolithic architecture, setting up an entire continuous delivery automation specific to this single service is not a big problem. But as new products emerge, or the product is decomposed into several microservices, or the same product starts to have different deployments for different customers, then the assembly of continuous delivery mechanisms starts to become a burden. This is when the platform becomes very convenient, as one of our interviewees lamented: *"We should have switched to a platform team years ago when we started having multiple products and multiple development teams*".

It was in the context of the company's decreasing focus on "tailor-made solutions for large clients" that a participant of our research revealed us that he had gained the board support (A24) for the construction of the platform: *"Multi-client products needed to be developed and produced more quickly. In this context, the platform became more important, to avoid the bureaucracy of the traditional environment*".

9.5 Consequences of API-mediated departments

A33 - Interaction patterns between devs and the platform team

The structure of API-mediated departments is much more about the "mediated" than the "API". This means that for everyday tasks, developers should not make requests directly, or via ticket, to infrastructure people. Developers must use the infrastructure autonomously, that is, using the automated platform. However, there are times when these teams must interact directly:

- *Resolving incidents together:* when difficult-to-resolve problems arise, teams work together to overcome the situation. However, the problem is initially handled only by the development team, which calls on the platform team only if necessary;
- *Providing consultancy to developers:* the platform is an internal product and, as a product, comes with support. Therefore, the infrastructure team spends time teaching platform best practices to developers (A37). Documentation and emails are approaches to alleviate individual tutoring;
- *Demanding new features:* Developers will at some point demand new features for the platform (for example, easy monitoring), just as customers generally demand the evolution of any software product. The platform team can also be proactive in probing developers' biggest needs through questionnaires or even conversations. One way or another, the infrastructure team takes care of the backlog (planned and prioritized improvements) of the platform.

Conflicts can also be considered a form of interaction. Of course, there will always be conflicts between interacting groups. However, conflict between API-mediated departments differs from those that occur in other scenarios. In segregated departments, conflicts often arise after an incident, with a focus on finding culprits. In a scenario

of collaborating departments, conflicts arise regarding the division of labor: it is not clear who does what. Finally, in an API-mediated structure, conflicts tend to precede problems in production, as they will be more around prioritization issues for improving the platform.

A real example of conflict was developers upset because the platform did not allow the installation of a certain library in the operating system, necessary for PDFs manipulation in Python. We are, therefore, talking about conflicts of the type *client versus supplier*, which occur for any software product.

A34 - The platform provides common mechanisms to handle non-functional requirements

In our research, we heard about various platform features, such as load balancing, auto scaling, rate limit, high-speed communications between data centers, security configurations, access to logs, provision of monitoring dashboards, etc. These are all useful resources for applications regardless of the domain; they should be used, or not, depending on the non-functional requirements in question. As one interviewee reported: *"Through the infrastructure modules we distributed, we facilitated the handling of this concern (...) other teams can use this same component, so that this high availability concern is considered"* . In other words, providing these facilities *" for free, effortlessly"* to the product team is at the heart of the platform.

Existing literature on platform teams is vehement about the need for platform being non-compulsory. The platform must be a product that developers wish to use, as it will make their lives easier, becoming a "compelling internal product" (BOTTCHER, 2018). Indeed, the incorporation of all these important resources to deal with NFRs is decisive for the platform to become a product that developers are eager to use.

A35 - Promotes continuous delivery

One of our interviewees commented the following: *"I understand that what we are doing as a consensus is the best to speed up and facilitate continuous delivery, to the extent possible, for infrastructure"*. As stated, one of the great points of the platform is to facilitate the deployment process. With that, different teams in the organization, with different levels of maturity, can achieve a certain degree of autonomy thanks to this facility.

The statement of another interviewee corroborates the importance of this autonomy: *"So the more autonomy you give to the teams, I think that firstly you go faster and secondly you scale better"*. We also remember that in addition to continuous delivery being the main objective of DevOps, being able to easily change the software at any time is also a great driver of the agile development mindset.

A36 - Devs are responsible for the infrastructure architecture

The options provided by the platform restrict the possible architectures for the applications deployed on it. However, within these limitations, there are still choices to be made and, although the platform team will eventually offer some advice, it is up to the product team to make these choices. In general, these choices correspond to meeting the non-functional requirements of the application. To cite two possible examples of choices provided by a platform: the number of replicas of a service and the distribution of that service across different data centers. An example of a restriction would be the non-availability of certain types of databases.

One of our interviewees called the restrictions imposed by the platform *guard rails* (which are literally protective walls used on highways), taking a positive view that this would help prevent developers from making certain mistakes. In fact, when using a third-party public cloud (e.g., AWS), the architectural restrictions on how to use the public cloud constitute one of the platform's greatest values. In this case, *constraints* also refer to the curation of public cloud services that can be used within the enterprise.

Sometimes, certain peculiar applications may have architectural needs not covered by the infrastructure platform. In general, the product team *"ends up changing the solution to fit this `[platform]` "* model. However, there is the possibility of allowing platform users to disable imposed restrictions if they wish. However, the ideal is developers to use these new requirements to demand some change to the platform. As one interviewee said: *"If many people use the same functionality, over time it is generally integrated into the platform"*. After questioning the real need for the change, the advantage is that, despite the wait for the requesting team, the new feature will benefit all development teams.

Unlike what tends to happen with segregated departments, developers in API-mediated departments are not limited to just coding. But, unlike single departments with developers with infrastructure background and attributions, here developers do not need deep infrastructure expertise to make the architectural choices made available by the platform. However, developers still need sufficient knowledge and responsibility to make architectural decisions. As one interviewee of our research said: *"There is an active effort to disseminate infrastructure knowledge among all devs; (...) we will teach everyone `[devs]` to have this as basic knowledge"*.

A37 - The platform team provides consultancy and documentation for devs

As with any product, the creators of the platform need to provide some type of support for that product. This ends up happening in a planned way (*"The infrastructure will become a platform and an internal consultancy"*) or not ("*Now, over time, I reached him, I looked at his code now as an internal consultant"* , a platform engineer told us).

Direct support in the form of consultancy is important but can become a bottleneck. Therefore, it is important that the platform team also provides quality documentation on the use of the

platform. The platform may emerge without good documentation, but its importance soon becomes evident: *"That's when this mentality began: not having to spend a lot of time explaining, you have to create good automation, you have to create documentation that can explain almost everything"*.

Of course, providing quality and up-to-date documentation is always a challenge, as a platform engineer warned us: *"So there came a time when, I don't know, 80% of the people on the `[platform]` team were focused on documenting things ... this made evolution quite difficult. (...) In fact, it greatly increased the dissatisfaction of the SREs themselves, having to do documentation all the time, dealing with small bureaucratic things (...), so this was kind of boring"*.

A38 - More devs do not require proportionally more infra people

This is a very valuable property of platform teams: the number of people needed to maintain the platform does not grow linearly depending on the number of teams served by the platform. Hypothetical example: let's say a platform team has 10 members and serves 10 development teams; To serve 100 teams, you may need a few more platform engineers, but just a few more, not 10 times more, which would require 100 people maintaining the platform.

This property occurs because the use of the platform does not depend on intense and daily communication between development and infrastructure areas (A33). The idea, according to one interviewee, is that *"if there were 10 thousand people, they would have to read the manual. By reading the API, they would know what to do"*. This does not happen for the model of collaborating departments or of single departments with dedicated infrastructure professionals, because, *"with each project that is born, with each team that is born, you will have to put someone there to operate together"*. This interviewee was emphatic: *"It wasn't working well to send ops people to the squads, because that doesn't scale"*. Among

the reasons for this conclusion, this interviewee commented: *"The ops guy takes time to understand how that squad works, how the product works, the project runs. And then there's a lot of turnover problem, so sometimes the squad is left without the ops guy, it borrows ops from another squad, then there are a lot of problems with that"*. Another interviewee concluded about API-mediated departments: *"I think it's basically the only mode that scales"*.

A39 - Eliminated previous bottlenecks

We have already seen that one of the motivations for adopting API-mediated departments is to overcome the bottleneck in the delivery path that was infrastructure management (A27). This is the same motivation as other types of DevOps transformations. However, according to our analyses, with API-mediated departments there is a greater chance of success (A35), resulting in greater delivery performance: *"He saw where the bottlenecks were, they were in the infrastructure. (...) it `[the platform ]` is actually helping developers bring value faster; (...) You don't need to ask for permission (...)"*.

One explanation for why this occurs is that API-mediated departments provide a better balance between specialization (division of labor) and coordination between teams: workers are more specialized than in the case of single departments, but interaction is more fluid than in the cases of segregated departments and collaborating departments. Additionally, the API-mediated approach reduces the costs of forming autonomous product teams by not requiring an infrastructure specialist on each of these teams.

A40 - Platform team as a small elite group

In general, the reports we heard were about platform teams with a relatively small number of people. One of them, for example, had 15 professionals in an IT company with more than 6 thousand employees. Another company with around 70 developers had an architecture "tribe" with 7 people, with the platform team included in this architecture tribe.

It is also common for the platform team to attract outstanding talent from the company. One interviewee expressed it this way about the technical superiority of the platform team: *"We `[SRE/platform]` are going to be a small closed team, there's no way to distribute these people, people messed up a lot in databases, things like that, we're going to have to take this obligation to deal with it out of their hands"*.

A41 - High costs using public clouds

Third-party public clouds (e.g., AWS, Azure, etc.) promise many facilities and low starting costs, which is convenient for startups with new products with low access volumes and no financial revenue. However, the question remains: what will the costs of these clouds look like in the future, after the success is reached? This cost ends up being an unintended consequence of platforms that are cloud facades when it is not taken into account in advance.

One participant of our research testified: *"We are scaling our Kubernetes project on bare metal `[without virtualization]` . And, for example, for monitoring services, if you have cloud services, it makes perfect sense to use SaaS. But for us it doesn't make sense, it's very expensive and doesn't offer the resources we need because we're on-premise."* Another interviewee also complained about the prices: *"I know that AWS queues are very expensive"*.

Although technological migrations are not simple, a possible precaution is worthwhile: when building a cloud-facade platform, build it as a specialized API for your company's needs, without leaking vendor-specific details to users. In this way, in the future it will be possible to migrate the platform from one public cloud to another, or even to an internal infrastructure, without this affecting the development teams.

Implicit interfaces

It's always a hope that we will be able to change the implementation behind an API without affecting its customers. But the bigger these changes are, the less chance there is of zero impact on these customers. A Google engineer even postulated a "law of implicit interfaces", which says that given a sufficiently high usage of an API, the maintainers of that API can no longer consider the implementation of that API as a private detail, unknown to customers; inevitably, together, clients will depend on every detail of the implementation, whether intentionally or not (https://www.hyrumslaw.com/).

However, this is no reason to neglect good abstractions. In addition to making the API easier to use, we hope that a good abstraction will reduce the impact on customers when there are significant changes to your implementation.

A42 - Dev skills very focused on the needs of the corporation

The automation brought by the platform brings many benefits to developers: ease, convenience, productivity, etc. However, some interviewees in our research saw a negative point for these people: learning about infrastructure is limited, important things are not learned. Examples of these gaps would be Kubernetes and Kafka technologies, which are widely used in the market and are often also the basis of internal infrastructure platforms, even if developers are sometimes not even aware of it.

This is not a problem for the company that employs these developers, but it can be a problem for these professionals when they want to change jobs, which is a common strategy nowadays to obtain a higher salary. Knowledge about an internal platform may not add much to the CV, while certain missing infrastructure knowledge may be a significant gap. In line with this reasoning, we learned from some developers who intended to leave a certain

company: *"Some devs wanted to leave precisely because they didn't know what they were consuming... The guy was so focused on the business logic that 'OK, I'm posting to a queue, but what's behind?'"*.

A43 - High cost to maintain the platform

We have already commented on how the use of a public cloud can make the operation of an internal platform more expensive (A41). However, whether it is a public cloud or your own, whether or not it is free software, it is necessary to consider the high costs of initial development (A25), maintenance, and operation of a platform. After all, even if internal, it is a product and should be treated as such.

One of our interviewees, working in a company with a platform but focusing on collaboration between departments, took this into account in his concerns about the API mediation model: *"The supplier-client model, in my view, is the most costly and the most dangerous... when you build tools like this, in a supplier-client model, you have a service, let's say, that has an operational cost, and I'm talking about money and time"*.

Therefore, a major tension that you must consider when building a platform is the shifting of costs between two alternatives: 1) operational effort when managing self-managed solutions — these can be free software, proprietary software, or developed by the company itself; or 2) financial cost when hiring a service managed by a third-party public cloud (example: Amazon EKS, which is Kubernetes managed on AWS).

A44 - Negligence of devs relying too much on the platform as magic

There is a risk in using an internal infrastructure platform: this platform may be seen as "magic" by developers, who end up neglecting quality because they trust the platform excessively so that they blame it for any problem without knowing what to do, even for simple problems or when the problem is in the application itself.

After all, *"you don't need to worry about how things work, they just work"*.

We've seen some issues with platforms that are too abstract (i.e., focusing on developer friendliness when trying to fully abstract the infrastructure). One of these problems is that developers expect too much from the platform. Sometimes the service is not running due to a coding or configuration issue, but developers inappropriately contact the platform team when they could have simply read the log file and discovered the problem by themselves.

As one interviewee told us: *"The deploy is magic, you click on a button and the application appears there. If I click on the button and it gives an error, I open a ticket and say: 'my deploy isn't working'. But why isn't it working? Do you understand how your pipeline works? Do you know what configurations you need to make?"* This problem leads to a question about how much the platform should abstract the infrastructure (A46).

A45 - Devs possibly unable to contribute to the platform

In the case of a small team (A40), it is common for the platform team to have some expectations about contributions to the platform from developers: *"I think a disadvantage of `[the platform]` would be that you don't give the developer a vision of what is happening. One way we think about it is to give freedom to the developer to know how we provide things, so they can be able to contribute to this model"*. In other words, the internal open-source model is used in the hope of bringing developers closer to the platform's implementation details.

However, even if developers do not contribute directly to the development of the platform, this openness helps to promote a relationship of trust: *"He `[dev]` feels part of the business because he can contribute to it `[ the platform]`, or if he is unable to contribute, he at least understands how it is being provisioned"*.

However, a warning is worth it. Knowledge of the platform's implementation details can increase the risk of coupling to these details, which in the future could make an internal restructuring of the platform more difficult (as discussed in A41).

9.6 Contingency for API-mediated departments

A46 - Decide how much devs must be exposed to the infra internals

The purpose of a platform is to abstract infrastructure so that developers can operate their services effortlessly. So, in principle, the greater the abstraction, the better. However, surprisingly, we observed at least two problems with very abstract platforms. The first is that developers expect too much from the platform and do not do their part (A44). The second problem is that to increase the abstraction of the platform, the platform team becomes more overloaded. Sometimes new functionality could be avoided if the devs wrote a configuration line. Even writing documentation is a factor for this overload (A37).

Therefore, the organization must carefully discuss the desired level of abstraction for its platform. We saw at least one case of a company in which this debate was promoted and there were several disagreements: *"The biggest argument was about the extent to which the dev has to understand the infra. I was one of the guys who radically thought that devs should know the minimum. But the guys who were tech leads thought differently, that devs didn't have to know anything, they didn't even have to know Docker"*.

Possibly, the organization will have to consider the skill level of its developers and also the expectations about their evolution; that is, whether being a better developer encompasses improving

operational skills, which will contribute to the reliability of services, or whether the focus is just on coding and "delivering soon".

KEY CHARACTERISTICS OF API-MEDIATED DEPARTMENTS

- The infrastructure team provides a platform with highly automated infrastructure services to empower product teams;
- Development teams and the platform team do not interact daily, but only in specific situations (incident handling, consultancy for developers, and demand for new platform features);
- The platform provides common mechanisms to handle non-functional requirements;
- Developers must operate their services;
- Platform engineers must have a product mindset and development skills;
- Require support from senior management through prior investment;
- Promote high delivery performance.

Capítulo 10
Enabler teams

An enabler team provides some type of support or facility for product teams, but without becoming responsible for maintaining business services or productive infrastructure.

The existence of these enabler teams was a theme that emerged repeatedly in our interviews. We heard about it in companies of the the four different DevOps structures. Therefore, we consider this phenomenon relevant to our research topic and, therefore, we briefly discuss the subject here.

Below we describe the most common types of enabler teams that we found in the companies that we analyzed in our research.

10.1 Consulting team

An enabler team can provide consultancy to developers on specific topics such as performance, security, and privacy. The most common type of consulting team we saw was the security team: even in a company with engineers with a more generalist profile, we found these specialists. Although enabler teams typically assist development groups, we also find security teams assisting infrastructure professionals.

Consulting teams can work in the audit model, listing problems to be solved by developers. In this situation, conflicts can also arise between these teams, especially when the development team is unable to prioritize the resolution of the pointed problems. These conflicts can be aggravated by the use of different vocabularies by these teams, as this makes understanding between them difficult.

In the audit model, there is still a choice that the organization must make: can the product be launched with pending issues identified by the consulting team? We have seen different approaches in different companies, but especially for the topic of security, there are expert group reviews that block the release until the problems highlighted are resolved.

In fact, we saw at least one case in which the security review was considered very rigorous. According to this interviewee: *"The preparation for the security review is not very obvious. They don't take it easy. The questions they ask... you have to be ready to explain the interaction between any two components very clearly. They really dive into the the details"*. Therefore, this blocking review scheme is expected to increase the security of the system, however at the price of introducing some additional delay in its launch.

Another question about the audit model is maintenance: when the launch consists of a small maintenance, the review can be waived. But in the case of slightly greater maintenance, it may be difficult to calibrate the need and rigor of these reviews. Consulting teams can contribute directly to the application code, but without becoming responsible for maintaining the code they produce. In other cases, they also provide tools, mainly libraries, to increase the productivity of development teams. In this context, in our research, we observed the availability of tools to facilitate the implementation of security, monitoring, high availability, automated tests, optimizations, and access to databases and messaging. The aim is to increase the autonomy of development teams through these tools.

In addition to consultancy focused on the systems being developed, the facilitating teams also seek to promote good practices among developers through lectures at the company.

Finally, a problem with this model is that, as the number of developers in a company increases, it becomes more difficult for a developer to get timely support from someone on the consulting team.

10.2 Deployment pipeline provider

In some companies, the "DevOps team" is a team dedicated to providing means for developers to have their pipelines easily and conveniently. One of our interviewees even stated that "*in Brazil, the DevOps position, without getting into the controversy, is about the people who create pipelines". In any case, the more microservices a team has, the more important it is to facilitate the creation of the pipeline for each service. In some cases, this may mean keeping a CI/CD (continuous integration / continuous delivery) system up and running, such as GitLab, for developers to use as they please.

There are different levels of autonomy that the development team can have about the team that provides the pipeline mechanism. The facilitating team may have to be called upon to create a pipeline for each service, which is undesirable in a microservices scenario, with new services constantly emerging. We saw a case, for example, in which the enabler team had to be called to disable static-analysis rules that the development team considered impertinent; our interviewee seemed annoyed with this bottleneck.

Another interviewee in our research valued his pipeline provider team for the gain in scale he saw in centralizing this type of service, concentrating a certain type of knowledge in this team while relieving the development teams of this work.

10.3 Coordination committee

In some cases, representatives from different areas (development, infrastructure, and possibly others, such as product and business) meet to make strategic decisions together in an aligned way, mainly about planning, delivery, and support. The idea is to bring together managers and technical leaders from each area who can give

opinions on the *"DevOps disciplines"* (planning, versioning, delivery, improvements, cloud, security, etc.). After the committee meetings, the impacted people take their *"homework"* to their respective areas.

In one of the analyzed cases, the coordination committee had a strong role in the security issue, demanding actions and imposing restrictions that improved the security of the solutions. As our interviewee said: *"The committee is instituting good practices, demanding more and more, placing more restrictions so that security flaws can be identified in the development* `[phase]` *"*.

In this case, it is worth highlighting that there are no people dedicated to the coordination committee since it is a group of representatives, who, therefore, must work in their respective areas. Thus, the committee functions as a coordination mechanism between areas.

Capítulo 11
Organizational structures beyond DevOps

The second part of this book discusses the structures used nowadays to organize development and infrastructure professionals in software-producing companies. However, the issue of organizational structures (division of labor versus integration between groups) goes far beyond software production.

This is a very special chapter, in which you will have the opportunity to understand how the challenge of organizing "devs and ops" is part of a much larger context. In other words, many of the conflicts between development and infrastructure professionals are not that specific. Here we will present a historical analysis that ends up unfolding into two other highly relevant topics. The first is a glimpse into *complex systems*, an emerging topic that relates to the dynamism of organizational structures. The second topic refers to a critical view of the concept of *culture*, a subject much talked about in the world of DevOps and agile methods.

This way, we hope you will have a broader perspective when dealing with conflicts between "devs and ops" in your organization. We also remember that the *lean* principles of software development had their roots in the precepts used by Toyota in automobile manufacturing. In other words, we do not advance in software production by thinking only about software, but also by reflecting on these other broader perspectives.

11.1 History of the organization in the industry

The roots of thinking about how the division of labor affects productive forces can be traced back to Adam Smith, who, in 1776,

placed three factors for productivity in the industry: 1) the use of machines; 2) the time savings when workers switch tasks (i.e., reduced context switching); and, precisely, 3) the increase in workers' dexterity resulting from their greater specialization (that is, the increase in the division of labor). Smith said that while a single worker could barely produce 20 pins per day, just 10 well-organized workers could produce thousands of the same pins per day in the same factory (SMITH, 1776). By "well-organized", we refer, using Smith's terms, to the "*division of labor*" and the "*combination of its different operations*", which is precisely equivalent to the concept of organizational structure used in this book.

Figura 11.1: On the 20-pound note (the currency of England), which bears the effigy of Adam Smith, one reads: "The division of labor in pin manufacture: and the great increase in the quantity of work that results."

Context switch

In the pin factories of the 18th century Scotland, the scenario exposed by Adam Smith, the manufacture of a pin had 18 steps, such as unwinding, straightening, and cutting the wire. Every time a worker changed tasks, time was lost. In the situation where only one worker would produce the pins alone, this loss would occur for all stages. In the scenario of well-organized workers, each worker was responsible for two or three tasks, so there was a saving in avoiding so many context changes.

In the pin factory scenario, the loss of time in context switching is due to factors such as employee movement and preparation of equipment and inputs for the next task. In the context of intellectual work, such as software development, context switching causes cognitive overload. In the case of software development, the main example of context switching is switching between different projects, but we can also mention, for example, the point at which the developer stops coding to wait for a battery of automated tests to be executed.

Due to the impact of context switching on software engineering, researchers have been investigating the subject. In a scientific article, for example, there is evidence that, in the scenario where developers work on many projects, those who focus on a few projects per day have greater productivity than those who switch between many projects per day (VASILESCU *et al.* , 2016).

Later, at the beginning of the 20th century, Frederick Taylor reinforced the view that the maximum division of labor would be the way to achieve productivity gains (OLIVEIRA, 2012). In this way, Taylor founded a school of management that advocates rational administrative procedures, clear channels of authority, centralized command and control, and interchangeability of employees. This system also values bureaucratic control: managers must avoid

favoritism, nepotism, and intuition. Taylorism also led to the emergence of Fordism, which focused on the mass production of standardized goods, initially automobiles, through the assembly line (JESSOP, 1992). This view of classical management laid the foundation for deeply segregating development and operations in the software industry.

Despite the notable popularity of Taylorist approaches until the 1960s, the truth is that the use of top-down rules and standards is only effective in stable environments (for example, on assembly lines). Furthermore, Taylorism worked with the assumption that global optimization would come from local optimizations, which is contested (see box *Local optimization vs. global optimization* in the *Impacts of DevOps* chapter). Furthermore, some industries have felt tensions between segregated departments. According to the documentary *21st Century Jet* by Karl Sabbagh, during the construction of the Boeing 777, for example, the previously isolated design and manufacturing departments were reconciled in the "working together" system, which promoted the free flow of information between departments and even suppliers. The ability to "work together" aligns with the DevOps movement's impetus for getting development and infrastructure teams to collaborate.

Yeatts and Hyten (1998) state that, in uncertain or unstable environments, employees need to make unprecedented decisions quickly, so that rules and standards matter less. In such a context, these authors argue that better performance is achieved with more informal and decentralized organizational designs. They then propose the adoption of self-managed work teams, which relates to the cross-functional team strategy. This position challenges the view that the maximum division of labor would be the supreme promoter of efficiency. Yeatts and Hyten define a self-managed work team as "*a group of employees responsible for managing and performing technical tasks that result in a product or service being delivered to an internal or external customer*". These teams would have the following properties:

- Typically have 5 to 15 employees;
- Be responsible for all technical aspects of the work;
- Periodically rotate technical and managerial responsibilities among its members;
- Not be groups with short-term purposes;
- Offer its members autonomy, responsibility, meaningful tasks, and learning.

When examining a few industries, Yeatts and Hyten found that self-managed teams achieve higher performance at lower cost and produce more innovation and creativity. The explanation is that decisions are more effective because the team members making decisions are the people most knowledgeable about the work. However, the authors recognize that it is difficult to rigorously prove such propositions.

The idea of self-managed teams impacted the software industry through agile methods. The Agile Manifesto, for example, explicitly states as a principle that "*the best architectures, requirements, and designs emerge from self-organizing teams*" (https://agilemanifesto.org/iso/en/principles.html).

In addition to maximum division of labor and self-managed teams, it is also possible for each employee to report to a product manager and a functional manager at the same time. This structure is called a matrix. One of our interviewees said that, in his company, the members of the multifunctional team officially reported each one to managers in their respective areas (development, infrastructure, and design); however, all members of this team jointly also reported to a person managing the product. According to this interviewee, this structure tries to avoid a problem with multifunctional teams, which is having a professional subordinate to – and being evaluated by – a manager who does not technically understand what this employee does. However, Henri Fayol (1916), one of the creators of classical management theory, states that this structure, by providing dual authority, is susceptible to confusion and conflict.

More sophisticated structures are also possible. The "front-back structure", for example, aims to achieve global scale (GALBRAITH, 2008). In this structure, one part of the organization (the front half) focuses on customers, while the other part (the back half) focuses on products, providing services to the front half. This resembles API-mediated departments (developers focus on customers and the platform team provides the back half of services).

Another very important alternative to Fordism and the maximum division of labor is the Toyota Production System (OHNO, 1978). In this system, mass production is restricted by a just-in-time philosophy (production on demand, avoiding stocks). Quality becomes a shared responsibility among all employees, not just a specialization: the production line must be stopped to correct problems. This philosophy influenced lean software development ideas, which focus on principles derived from the Toyota Production System, such as reducing waste and bottlenecks (POPPENDIECK, M.; POPPENDIECK, T., 2006).

Lessons from history for our present and future

An important lesson from this section is how segregated department structures did not arise without reason. Sometimes people in the DevOps world may be tempted to despise organizations with a tradition of silos. But empathy is needed, as this structure was built on a long, centuries-old, tradition of organizing industry with a view that efficiency comes from maximum division of labor. At first, it makes sense: a professional who dedicates himself solely to coding (which already encompasses a world of learning) would be an increasingly better coder, not having to worry about tasks outside his expertise.

Understanding that autonomous teams are necessary for greater organizational performance depends on a deep reflection on the essence of the software. This means understanding that the factory metaphor does not apply so well to software production, mainly because it works with a low level of repetitiveness (typically, each

new project is a new learning process). However, when hiring the development of a system, people still have a strong impulse to decide on the scope, price, and deadline in advance, which reveals a mistaken understanding of the essence of the software.

Here it is also necessary to understand that the structures of single departments and API-mediated departments are different strategies that seek to form autonomous teams (while single departments seek to maximize autonomy, API-mediated departments compromise a little autonomy in exchange for a greater specialization of the workforce). Therefore, when faced with future possibilities for organizing teams, you already know that you must evaluate the impact of these options on the autonomy of the professionals involved.

11.2 Complex systems and the dynamism of structures

It is also possible to analyze an organizational structure beyond the dimensions of division of work and integration between teams. There are, for example, studies that are more concerned with the dynamics of organizational changes than with understanding the structures at a given point in time. Such studies are anchored in systemic thinking, related to complex systems, which is a dynamic view of organizations based on biological metaphors (MORGAN, 2011). Complex systems consider the interaction between growth processes and limiting processes. Senge *et al.* (1999), for example, identify growth processes – "because it matters", "because my colleagues take it seriously" and "because it works" – and strategies to limit these processes – for example, "don't push so hard for growth".

Researchers also represent complex systems as networks of factors that affect each other through reinforcing and balancing feedback

loops (PENZENSTADLER, 2018). Furthermore, complex systems include leverage points, in which small changes in one aspect can trigger significant changes throughout the system – the famous “butterfly effect”. In other words, while classical management focuses more on the division of labor between work groups, complex thinking focuses more on the integration of such groups: "*how things connect are more important than what they are*" (SINCLAIR, 2020).

Assuming that organizations are complex systems has implications. Sosa *et al.* (2004) found a strong tendency for alignment between system interfaces and interactions between teams, corroborating Conway's Law. Still, in complex systems, they realized that such matching is not perfect. Another implication is that the central processes of organizations must focus on change (and not on how to always do things the same way), as Toyota's *katas*, which are work patterns or routines focused on continuous improvement and adaptation (ROTHER, 2009).

Conway's Law

Conway's Law compares graphs abstracting system structures (i.e., interconnections of subsystems) and structures within an organization (i.e., communication paths between groups of people) (CONWAY, 1968). An example reported by Conway himself is the construction of compilers, in which the number of compiler phases does not derive from intrinsically technical aspects. In the example, a compiler developed by three people ended up structured in three phases, while another compiler developed by five people ended up having five phases. In other words, the structure of the software mirrors the structure of the people or teams that produce that software. This is very important in the context of this book, as the notion of structure used by Conway corresponds to the concept of organizational structure that we employ.

Nowadays, Conway's Law has gained renewed importance for the development of systems adopting microservices architecture. In this context, the main motivation for splitting an existing service into two is to prevent the same service from being maintained by two different teams. Therefore, breaking a system into microservices seeks to adapt the system architecture to Conway's Law.

Authors who study complex systems have also published books to help managers make organizational changes in their companies. They list elements necessary for successful transformations, such as support from the CEO and reduction of barriers within the organization (SENGE *et al.*, 1999). Kotter and Cohen provide eight steps toward change: "*increase urgency*", "*build the guiding team*", "*get the vision right*", "*communicate for buy-in*", "*empower action*", "*create short-term wins* ", "*don't let up*" and "*making change stick*" (KOTTER; COHEN, 2012).

Manns and Rising argue that change is best introduced bottom up with support at appropriate points from management (MANNS; RISING, 2004). They also see change and innovation as processes (and not as events), consisting of stages: knowledge, persuasion, decision, implementation, and confirmation.

Niels Pflaeging (2014) has a proposal to structure production in complex organizations towards high performance. He advocates a drastic change in management approach to a "non-managerial" approach, organizing the company into a value-creating network of self-managed teams that interact as peers. It also proposes broad employee participation in decision-making, roles rather than titles for employees, informal leadership, and evaluating team performance rather than individual performance.

Davis and Daniels (2016) call this structure *holacracy*, in which decisions are made by self-organizing teams, not following a traditional management hierarchy. They also warn of the risks of holacracy, in which career development is less clear, and people without the necessary management skills end up unofficially filling power vacuums.

In this book, we present different organizational structures and some advice about how you can help your organization to undertake a transformation towards one of these structures through their causes and conditions. However, delving deeper into the topic of complex systems is an excellent way to better prepare yourself to lead an organizational change process.

Delving deeper into the topic also brings fundamental revelations about the essence of software systems. Complexity concerns how complex and elegant behaviors of the system emerge from simple rules of each of its components (FRIED *et al.*, 2012). It's the famous "the whole is greater than the sum of its parts", which applies well, for example, to systems based on microservices. But for beautiful complex patterns to emerge from your systems, you need to focus

on simplicity (that is, avoiding complicated rules) in each component of the system.

A final example of concept related to complexity in the context of organizations, but which also applies to software systems, is *scaffolding*, which denotes structures created to support changes, but in such a way that these structures are discarded after the change (SINCLAIR, 2020). An example of scaffolding for systems is the use of feature toggles (a configuration that activates or deactivates a functionality without the need of deployment). An example of organizational scaffolding could be a DevOps team that seeks to instill a new culture in the organization. In fact, let's talk about culture now.

11.3 Culture: the most important thing?

A significant portion of the business books we mentioned in the previous section see action on culture, people's feelings, and emotions in the corporate context as more important than action on structures. Would our concern with organizational structures be negligible? Let's analyze.

Senge et al. (1999) state that "*it is not enough to change strategies, structures, and systems unless the thinking that produced these strategies, structures, and systems also changes*". Kotter and Cohen (2012) reported: "*The central issue is never strategy, structure, culture, or systems (although these elements can be very important). The core of the matter is always about changing the behavior of people*", which is contradictory, as they define culture as a group of norms of *behavior* and shared values. Manns and Rising (2004) relate culture to the acceleration and deceleration of innovation.

These ideas about the precedence of the cultural factor have been echoed in the software community. For example, an interviewee in our research, a consultant, even made a statement similar to this vision: *"Will the devs who are there have this agile mindset? If not, it's pointless"*. In the book *Effective DevOps* (DAVIS; DANIELS, 2016), we also find statements like that, such as the definition that effective DevOps would be "*an organization that embraces cultural change to alter the way individuals think about work*", or stating that "*DevOps is a prescription for culture. (...) DevOps is about finding ways to adapt and innovate social structure, culture, and technology together to work more effectively.*" Furthermore, the "third way of DevOps" from the *DevOps Handbook* is related to improving daily work through a specific type of culture (KIM et al., 2016). In other words, these authors argue that culture affects practice.

But... What is culture?

The book *Effective DevOps* defines **culture** as "*the ideas, customs, and social behavior of a people or society*". Still in the DevOps context, the book *Accelerate* (FORSGREN, 2018) adopts a culture model focused on how information flows through the organization. In this context, a culture can be pathological (power-oriented, involving fear and threats), bureaucratic (rule-oriented), or generative (performance-oriented).

However, this precedence of culture is debatable. Consider the view of Pflaeging (2014), a scholar of complex organizations and systems:

"Culture is not a success factor, but an effect of success or failure. (...) Therefore it cannot be influenced directly. (...) Culture is observable, but not controllable. (. ..) A company cannot choose its culture, it has exactly the culture it deserves. Culture is something like the restless memory of an organization (...) It provides something like a common 'style' that everyone can trust (. ...) Culture, in this sense, is autonomous. Culture is neither a barrier to

change nor does it encourage change. (...) Culture allows us to indirectly observe the quality of change efforts: change trickles through into the culture."

We understand that this view is corroborated by other positions, including historical materialism, proposed by Karl Marx, for which the mode of production (we can include its structures) of society conditions the superstructure of that society (which includes culture): "*The mode of production of material life* `[the infrastructure]` *conditions the general process of social, political and intellectual life* `[the superstructure]` *(...) Changes in the economic infrastructure lead sooner or later to the transformation of the whole. the immense superstructure*" (MARX, 1859).

Back in the context of software production, Forsgren *et al.* (2018) state that "*it is possible to influence and improve culture by implementing DevOps practices*". They found that lean product development (e.g., working in small batches) and management (e.g., limiting work in progress) practices promote a performance-oriented culture. In the agile world, it is common to say that agile is about culture, not practices or tools. However, Robert Martin, *Uncle Bob*, states that "*you cannot have a culture without practices; and the practices you follow identify your culture*" (MARTIN, 2014). In other words, these authors understand that, above all, it is practices that affect culture and not the other way around.

Consider a company that shifts the assessment process from individual-centered to team-centered. The evaluation process is part of the production system, and will certainly have a major impact on the way people behave and interact. Organizational structure can also influence behavior. Consider having a QA (quality assurance) team. The presence of this structure can encourage developers to not be as careful in their testing, possibly generating applications with more defects.

In the DevOps context, holding developers responsible for infrastructure management can promote a culture of learning (there

are no people to push problems to). On the other hand, trying to establish a culture of collaboration between developers and infrastructure personnel (previously working in silos) without changing structures can cause conflicts due to unclear responsibilities.

Therefore, we believe that caution is needed with initiatives focusing only on culture, or considering that cultural change must precede other important changes. At the very least, before trying to change a culture, it is essential to investigate the reasons that led people to adopt a certain culture, that is, to behave in a certain way.

In this chapter, you could see that the general literature on organizations and their structures is vast and rich. However, you may also notice that these studies are not consensual in many aspects, which requires a certain amount of critical sense of whoever is reading. Finally, we hope that the issues discussed in this chapter will provide you with a broader perspective to interpret and apply the propositions of this book, in addition to serving as a seed for you to continue studying organizations.

Capítulo 12
Conclusions

The workforce dedicated to software production has grown across the world. However, customer demands have also increased: new features must be delivered faster and faster. Furthermore, new features must support more users while dealing with more demanding non-functional requirements (for example, society and governments are now demanding more security and data protection in software products).

In this scenario, it is not uncommon for companies to try to meet such demands by simply putting more pressure on their employees or expecting a single employee to master a vast set of skills. Likewise, it is also not uncommon to experience software slowdowns or unavailability in our daily lives. Addressing these issues is not just a matter of demanding more from each employee. As stated long ago by Adam Smith (1776), while a single worker could barely produce 20 pins per day in a factory at the time, just 10 employees, when well organized, could produce thousands of pins per day in the same factory.

To face the contemporary challenges of software production, we hope that advances brought by software engineering research, including ours, can promote more rational ways of organizing the software production workforce. In particular, we hope that such progress will benefit companies (who will produce more software), workers (who will have better working conditions), and consumers (who will enjoy better software).

Let's now have a brief recap of the ways to organize DevOps and some practical advice resulting from this. Finally, we offer a small list of recommended readings, accompanied by mini-reviews, on major works on the DevOps topic.

Revision

With this book, you learned about how software companies have been organizing their development and infrastructure workforce according to different organizational structures. These DevOps structures are: segregated departments, collaborating departments, single departments, and API-mediated departments. The following table summarizes these structures, presenting for each of them: *(i)* the division of work between the development and infrastructure groups in relation to operational activities (e.g.: deployment and incident handling); and *(ii)* how these groups interact with each other.

Organizational structure	**Development team**	**Infrastructure team**	**Interacti**
Segregated departments	Only generates the application deployable package	Responsible for all operational activities	Limited collaborati between groups
Collaborating departments	Collaborates with some operational activities	Responsible for all operational activities	Intense collaborati between groups
Single departments	Responsible for all operational activities	Does not exist	-
API-mediated departments	Responsible for all operational activities, but with the	Provides the platform, automating much of the operational activities	Interactior happens ir specific situations, not on a daily basis

Organizational structure	Development team	Infrastructure team	Interacti
	support of the platform		

You may also have understood that it is difficult to talk about "the best way to organize DevOps", because, like everything in software engineering, it depends on the context. In particular, each way of organizing DevOps has different conditions, causes, reasons to avoid, consequences, and contingencies. In summary:

- Along with the description of each DevOps structure, you saw why different organizations adopt (or not) different structures and the conditions that lead to this choice. For example, one cause for API-mediated departments is trying to resolve delivery bottlenecks; one reason to avoid single departments is their unsuitability for applying corporate standards; and a condition for collaborating departments is to have a certain ratio between infra people and development teams.
- We were also able to explore the disadvantages of each structure and how organizations deal with these disadvantages. For example, one consequence of collaborating departments is that they possibly lead to conflicts due to unclear responsibilities. Calibrating the platform's level of abstraction is a contingency to prevent developers in API-mediated departments from relying too much on the platform as something magical.

Delivery performance

While reading the book, you were able to understand the relationship between different ways of organizing DevOps and delivery performance. While we have evidence that API-mediated departments provide better delivery performance, we have seen that segregated departments are more common in organizations that do

not have high delivery performance. An interpretation of these relationships can be made considering two classic productivity factors: 1) the degree of specialization of workers and 2) the number of context switches during the process (SMITH, 1776).

In segregated departments, we have high specialization in search of optimizing "resources". However, transferring work between areas, normally via the ticket system, involves a large amount of context-switching for development professionals who interrupt their workflow when opening a ticket to the infrastructure area.

Collaborating departments seek to reduce wait times and thus also reduce context switching through direct collaboration between development and infrastructure professionals. Here the hope is that a new culture of collaboration will make a difference. But without a sufficient number of infrastructure professionals available to collaborate, which happens, no culture will work miracles: while an infrastructure professional collaborates with a developer, the other developer must wait. Furthermore, the collaboration system reduces specialization, as some responsibilities are poorly defined.

In single departments, the attempt is to eliminate waiting time by making a single group responsible for both development and infrastructure. However, this reduces the specialization of the group developing the software in the first place. Second, although the team does not need to wait for the infrastructure center to resolve the ticket, it is now the team itself that must switch contexts (from "dev mode" to "infra mode") and invest this time in infrastructure issues.

Finally, the API-mediated structure seems to provide a good balance between specialization and reducing context switching. Specialization remains: the infrastructure team provides a platform encapsulating infrastructure knowledge, while developers develop the application. At the same time, context switching is low for developers, as they operate their services autonomously through the infrastructure platform. Here, we consider that the platform provides an abstract

and easy way for developers to operate the infrastructure, which reduces the cost of forming autonomous teams.

Note that, for API-mediated departments, specialization and context switching reduction are not maximized. Each team focuses on a specialty, but developers have to understand a minimum of infrastructure to operate the platform. The platform team needs some development mindset and skills since they are offering the platform as a product. In other words, there is less specialization than in segregated departments, but more specialization than in single departments. At the same time, the developer must stop his coding and testing work to operate the service through the platform, even if the platform seeks to facilitate this operation as much as possible. Therefore, there is, at the very least, less context switching than in segregated departments.

In summary:

- Segregated departments favor specialization, but increase context switching;
- Collaborating departments try to reduce context switching, but with some reduction in specialization;
- Single departments reduce context switching, but harm specialization;
- API-mediated departments provide a better balance between specialization and context switching.

In time, you must have some critical sense to understand that achieving high levels of delivery performance is not a necessity for all software-producing organizations, just as the adoption of API-mediated departments is not suitable for every company.

Practical advice

The practical application of the content in this book is highly context-dependent. That's why you must have a critical sense to interpret the reality of your organization in light of our propositions and thus

find a healthy path to promoting a DevOps transformation that brings effective results. Taking this into account, and based on what we studied in this book, here is some practical advice that you should consider:

- If, for any reason, your organization needs to strongly segregate development and infrastructure professionals, do not make high delivery performance an organizational objective.
- If your organization doesn't have enough infrastructure staff to collaborate freely with developers, don't try to establish a strategy of collaborating departments.
- When adopting collaborating departments, advise professionals that it is not feasible to predict the detailed responsibilities expected for each role. Instead, actions must be taken based on aligned objectives between departments.
- If you want to have highly autonomous teams that can move quickly, be ready to give up imposing corporate standards on the infrastructure of these teams. Additionally, consider some measures so that each team is knowledgeable about infrastructure: hire enough infrastructure staff, embrace cloud automation, and give workers time to improve their skills.
- If you want to achieve high delivery performance by adopting an in-house infrastructure platform, be prepared to pay the price upfront: prepare infrastructure personnel with development skills and increase developers' responsibilities for operations and non-functional requirements.
- Define the platform's expected level of abstraction and let people know about that expectation. This measure can alleviate conflicts and smooth collaboration between developers and the platform team. Also, advise the developer and platform teams about the expected interaction patterns.

Recommended readings

We can find hundreds of books and articles about DevOps out there. We know it's simply impossible to read all of this! So, which reading

to prioritize? Here we present a small list with suggestions with some books that we think are the most relevant.

Continuous Delivery **(HUMBLE; FARLEY, 2010):** This book is one of the foundations of the DevOps movement. He is the one who initially presents the anatomy of a deployment pipeline, covering various technical aspects of software deployment, a phase typically neglected in more traditional software engineering books. The book values automation as a fundamental element of a fast and reliable delivery process, at a time when manual (and often problematic) deployments were the norm. It is also in this book, obviously from the title, that the idea of continuous delivery is presented — that is, the idea that each commit must be potentially deployable. The book also addresses issues such as configuration management, version control, test automation, database evolution, and environment management, all vital subjects for the success of a software system in constant evolution.

Accelerate **(FORSGREN *et al.*, 2018):** Using rigorous research methods and the application of surveys with 23 thousand respondents from 2 thousand different organizations, the book presents the factors that accelerate software delivery. In particular, the authors explore which technical practices contribute to continuous delivery. But more importantly, the book reveals that constant software delivery positively impacts very important organizational results: profitability, productivity, market share, effectiveness, efficiency, and customer satisfaction. It is in this book that the authors present a proposal on how to measure delivery performance, enabling respondents to be classified into low, medium, and high-performance contexts.

The goal **(GOLDRATT; COX, 2014):** This is a book from the 1980s that tells the story of Alex Rogo, manager of an industrial plant. The story begins in a chaotic atmosphere in which an attempt is made to avoid delaying an important delivery while the company threatens to close Alex's unit. He will then have to save the jobs of their colleagues. To do this, he will count on the help of Jonah, a

former physics professor, who will guide him on a journey based on the Socratic method: Alex will not have ready answers, but he will have the crumbs that will show him the path to important discoveries. The most fundamental revelation of the book is about how to manage bottlenecks: there is no point in optimizing steps that are not the bottleneck in the process. It seems obvious, but in the manufacturing context of the book, there was a common obsession with accumulating parts (creating stock, which has negative consequences) to minimize the total time spent preparing machines. At points that are not global bottlenecks, this attitude does not improve the overall performance of the system; on the contrary, the book shows that batches should be reduced for steps that are not bottlenecks. These ideas also underlie lean software development.

***The Phoenix Project* (KIM *et al.*, 2018):** This book tells the story of Bill Palmer, an IT operations director in a large industry. The story begins in a chaotic scenario with ongoing high-severity incidents. To solve the company's problems, the CEO determines the launch of the Phoenix Project in a few weeks; detail: the project has already been delayed for years and has already consumed millions more than planned. But Bill will have to guarantee the success of the project, otherwise the company's IT will be outsourced in case of failure. To do this, Bill will have the help of Erik, an eccentric member of the board, who will explain to him about the Toyota Production System and lean principles. Bill will then apply the three ways of DevOps, created by one of the book's authors and presented in the also classic *DevOps Handbook* (KIM *et al.*, 2016), to guide his colleagues to success through collaboration in the organization. And yes, you may have noticed, the book is heavily based on the previous one, *The Goal*, translating into the software context concepts like inventory (example: undeployed code). One of these revisited concepts, also present in the book *Accelerate*, is about the importance of some level of worker idleness to avoid long waiting times when transferring work between units.

If you want to delve deeper into platform teams, these are some of the most relevant readings that introduced the topic:

- *Platform engineering product teams*. Available on Thoughtworks Technology Radar (2017-2021): https://www.thoughtworks.com/en-us/radar/techniques/platform-engineering-product-teams.
- *What I Talk About When I Talk About Platforms*. Post by Evan Bottcher (2018) on Martin Fowler's blog. Available at: https://martinfowler.com/articles/talk-about-platforms.html.
- *Team Topologies*. Book by Matthew Skelton and Manuel Pais (2019), published by IT Revolution.
- *State of DevOps Report 2020*. Available at: https://circleci.com/resources/state-of-devops-report-2020/.
- *Mind the platform execution gap*. Post by Cristóbal García and Chris Ford (2021) on Martin Fowler's blog. Available at: https://martinfowler.com/articles/platform-prerequisites.html.

In addition to books and blogs, scientific articles are another information source you can consider. Our literature review (LEITE *et al.*, 2019) points to a list of 50 good articles already published on DevOps. To follow new publications, you can create an alert on Google Scholar to receive notifications in your email about new articles published on DevOps or another topic of your choice. If you are not used to this type of reading, the following appendix will help you get an idea of the essence of academic production.

Finally, we would like to thank you immensely for reading this far. We hope that this book has provoked reflections accompanied by new perspectives on this collective endeavor that is software production. We also hope that these reflections and perspectives will be of value to you throughout your career. So, we end by wishing you success!

Capítulo 13
Appendix — Our scientific approach

The content of this book is the result of scientific research carried out at the University of São Paulo (USP). In this chapter, you will understand a little about the world of academic research and how we conducted our research on DevOps.

Research and scientific method

In most of the countries, scientific research is usually carried out by university professors together with master's and doctoral students. In general, the objective of research is to produce knowledge, that is, to make some discovery about how the world behaves or even to bring a new perspective on some phenomenon. Thus, research is typically built around a *research question*. In our case, this question would be about "how has the software industry been organizing development and infrastructure professionals?" But the way to answer a research question cannot be done in any way. It is necessary to rigorously apply some scientific method.

Yes, that's right, "some method". What we normally conceive as the scientific method corresponds to the following flow: a scientist has an idea and formulates a hypothesis from it (a proposition that can eventually be proven false); formulates an experiment to test the hypothesis; runs the experiment; reaches conclusions about the hypothesis. If, in addition to being able to predict some phenomenon (when A happens, then B occurs), the scientist has a good explanation about why it occurs (when A happens, then B occurs, and this is because of XYZ), we can say that she has a theory in hand. However, not all science is done this way.

Theory

Broadly speaking, a theory is a system of ideas to explain a phenomenon (RALPH, 2019) or knowledge accumulated systematically (GREGOR, 2006).

First, where do hypothesis ideas come from? Everyone knows the story of the apple that fell on Isaac Newton's head. But we cannot expect all scientists to be this enlightened. Therefore, there must also be methods to support the conception of hypotheses. Furthermore, in many fields of study, it is very difficult, or almost impossible, to conduct controlled experiments. This is the case, for example, of social sciences: the laboratory of social sciences is history, what has already occurred; scientists can't control sociological events.

Software engineering research

Software engineering is a field of research that studies the various aspects of software production. Because software is made by groups of humans, software engineering has a lot of social sciences. This is what we see in this book: how different people, belonging to different human groups, organize themselves to carry out a process. Thus, software engineering also shares the methodological difficulties of the social sciences.

EXAMPLES OF SOFTWARE ENGINEERING RESEARCH

Here are some of the topics presented in the 2022 edition of the most important software engineering conference in the world, ICSE (*International Conference on Software Engineering*):

- Coordination of remote teams.
- Perception of productivity in software teams.
- Privacy in mobile applications.
- Code summarization (i.e., generation of text that explains a source code).
- Automatic detection of performance problems in databases.
- *Bots* (robots) analyzing *pull requests*.
- Collaboration in the construction of systems based on machine learning.
- Toxicity in open source communities.
- Testing deep learning libraries.
- Use of *cannabis* combined with programming activity.

The complete list can be accessed in the ACM (Association for Computing Machinery) digital library: https://dl.acm.org/doi/10.1145/3510003.3510156.

Therefore, software engineering uses different scientific approaches to achieve its results. And one of these approaches in vogue is *Grounded Theory*, which we will explain below.

Grounded Theory

The scientific methodology that guided our research process is called ***Grounded Theory*** (GLASER; STRAUSS, 1999). This methodology emerged in the social sciences, in the 1960s, to support researchers in formulating theories based on qualitative data in a disciplined way, since, at that time, only quantitative research (using numerical measurements) was taken seriously, even in the social sciences. The

term *grounded* means that a theory produced under this process is grounded on (is based on) data (even if qualitative).

The "constant comparison method" is the central technique for producing data-informed theory. It is based on rigorous analysis of qualitative data (in the case of our research, interview transcripts) and is realized through "coding", a process of condensing the original data into a few words with conceptual relevance. When writing down a concept, the researcher must compare its occurrence with previous occurrences of the same or similar concepts. When coding, if conflicts and reflections on theoretical notions arise, the researcher should write down a memo about these ideas. A memo is an unstructured note that reflects the researcher's thoughts at a specific point in time.

Data collection must follow a theoretical purpose so that the researcher can define groups whose comparison is interesting. Data collection and analysis are interleaved and complement each other so that the emerging theory guides which data should be collected next, considering gaps and questions suggested in the analysis carried out so far. This process is called "theoretical sampling."

The researcher must continue the analysis until "theoretical saturation" is reached, which means that new data no longer significantly impacts the formulated theory. However, as a constantly evolving entity, not a finished product, new data can always be analyzed to alter or expand a data-driven theory. Consequently, practitioners can adjust the theory when applying it to their concrete scenarios.

Our research process

First of all, to find a good research question that was relevant and not yet well answered by the existing literature, we reviewed the most important articles already published on DevOps. The results of this literature review correspond to the first part of this book.

After reviewing the literature, with a research idea in mind, we initially spoke to seven DevOps experts to get feedback on our ideas, and we were able to verify that we were on a promising path. We, therefore, aimed to understand the different ways of organizing development and infrastructure professionals in software-producing companies. These initial conversations also provided us with the background to frame the interviews to come.

From there, the research was structured in two phases. In the first phase, the objective was to discover the different ways of organizing DevOps currently in use, providing a *description* of these possibilities. To do this, we spoke to 37 professionals in the software industry. Following the idea of theoretical sampling from *Grounded Theory*, our idea was to interview different types of people working in different types of companies. We then interviewed professionals from different roles, genders, and experience levels, working in companies of different domains, sizes, and countries. To get an idea, we interviewed people working in Brazil, the United States (USA), Germany, Portugal, France, Canada, and Italy, as well as professionals working in globally distributed teams.

The main topic of these conversations was the actors involved in the different operational activities of software systems (for example: incident handling). As a result of the first phase, we obtained a taxonomy (classification) of the different DevOps structures and their associated options.

Then, in the second phase of the research, the objective was to better understand these structures, comprehending their advantages, disadvantages, and application contexts. To do this, we interviewed another 31 workers. At that moment, the point was to interview people working in the different DevOps structures already identified. We also tried to balance the number of development and infrastructure people. Another relevant point is that we interviewed both people in companies already analyzed in the previous phase and new companies. At this stage, we spoke to people working in Brazil, the USA, and Spain.

The second phase of interviews was analyzed using a type of coding that sought to identify conditions, causes, reasons to avoid, consequences, and contingencies for each DevOps structure. The interview questions reflected these aspects but analyzing the specific case of each interviewee. The result of this was the creation of an *explanatory* theory about the analyzed phenomenon.

Furthermore, as the conversations in the second phase already used the terms from our taxonomy, we took the opportunity to check whether these terms were actually facilitating the conversation or just causing more confusion. Based on this analysis, we were able to refine the terms of our taxonomy, finding more suitable terms to express our ideas.

At the end of each phase, we also sent questionnaires so that the people interviewed could give their opinions on the results obtained. Overall, we received positive feedback.

Publications: thesis and articles

The explanation of our research process in the previous section was quite summarized. Details of all our procedures can be found in the doctorate thesis *A grounded theory of organizational structures for development and infrastructure professionals in software-producing organizations* (LEITE, 2022), available at the USP Digital Library of Theses and Dissertations at the following address: https://www.teses.usp.br/teses/disponiveis/45/45134/tde-28062022-132626.

In fact, in scientific writing a great focus is required on the description of the research process used, even more than on the results obtained: it will be by its method that a research will be mainly judged. Therefore, there is a big difference between the thesis and this book: the book highlights the results of the research, and not the employed method. Here in the book, the results are even more detailed and consolidated, while in the thesis they are

presented in parts corresponding to the research phases (a history of the theory evolution, not its final state).

Another difference between the thesis and the book is that in the thesis we are much more concerned with comparing ourselves to other existing studies and highlighting the novelty of what we bring. For those who missed it, in the thesis there is a brief comparison between our work and the book *Team Topologies* (SKELTON; PAIS, 2019), a famous publication that deals with topics similar to our research. One of the highlights of *Team Topologies* is its discussion of platform teams. Nonetheless, we highlight here that *Team Topologies* was published during the development of our research, and that one work is not based on the other.

In short, in scientific writing, the paradigm is that of distrust: the reader must be convinced that your work is important, that you did everything correctly, and that your work brings something new in terms of knowledge; that is, you must prove that your work deserves to exist. For a book like this, the paradigm is convenience: the reader is someone who wants to learn something, and the author seeks to be as didactic as possible so that the reader has a pleasant read and fulfills her learning objective.

In academia, there is also the publication of scientific articles, which is the main way in which scientific results are disseminated, thus becoming one of the major objectives of every researcher. These articles are published in scientific journals or presented at conferences (also called symposiums). The great value of this type of publication derives from its rigorous approval process since each article is subject to peer review, which means that the evaluators are other researchers in the area of the submitted article.

In our case, we were fortunate to have three articles about our research accepted by renowned high-impact international journals:

- The article *A Survey of DevOps Concepts and Challenges* (LEITE *et al.*, 2019), about the literature review on DevOps, published

in the *ACM Computing Surveys*;
- The article *The Organization of Software Teams in the Quest for Continuous Delivery: A Grounded Theory Approach* (LEITE *et al.*, 2021), about the results of the first phase of our research, published in the *Information and Software Technology* journal;
- And the article *A theory of organizational structures for development and infrastructure professionals* (LEITE *et al.*, 2023), about the results of the second phase of our research, was published in the *IEEE Transactions on Software Engineering* journal.

If you would like to contribute (that is, making discoveries) on topics related to software engineering, consider the possibility of pursuing a master's or doctorate. If this is the case, talk to students and professors at a university. At IME-USP (Institute of Mathematics and Statistics of the University of São Paulo), for example, we have an applied research group in the Department of Computer Science that researches the areas of software engineering (with a great tradition in agile methods and free software), distributed systems, smart cities, databases, parallel computing, and networks. Also, feel free to contact the authors of this book about engaging yourself in the research world. :)

Capítulo 14
Bibliographic references

For more convenient reading, in this book we have omitted several references used in our work. For more references used in our research, please consult the works listed here authored by LEITE, Leonardo.

21ST CENTURY Jet: The Building of the Boeing 777. (Series). Directed and produced by: Karl Sabbagh. United States: Skyscraper Productions, 1996. 4h45min.

ALLSPAW, John; HAMMOND, Paul. *10+ Deploys Per Day*: Dev and Ops Cooperation at Flickr. 2009. Slide presentation. Available at: https://pt.slideshare.net/jallspaw/10-deploys-per-day-dev-and-ops-cooperation-at-flickr. Accessed in: 11 Oct. 2023.

BAI, Xiaoying; LI, Mingjie; PEI, Dan; LI, Shanshan; Ye, Deming. Continuous Delivery of Personalized Assessment and Feedback in Agile Software Engineering Projects. In: 40th International Conference on Software Engineering: Software Engineering Education and Training, Jun. 2018, Gothenburg, Sweden. *Proceedings...*, 2018. 10 p.

Available at: https://netman.aiops.org/wp-content/uploads/2018/03/ICSE-SEET-36.pdf. Accessed in: 11 Oct. 2023.

BASIRI, Ali; BEHNAM, Niosha; DE ROOIJ, Ruud; HOCHSTEIN, Lorin; KOSEWSKI, Luke; REYNOLDS, Justin; ROSENTHAL, Casey. Chaos Engineering. In: *IEEE Software*, v. 33, no. 3, p. 35-41. 2016. Available at: https://ieeexplore.ieee.org/document/7436642. Accessed in: 11 Oct. 2023.

BEYER, Betsy; JONES, Chris; PETOFF, Jennifer; MURPHY, Niall. *Site Reliability Engineering:* How Google Runs Production Systems. Sebastopol (EUA): O'Reilly, 2016.

BOTTCHER, Evan. What I Talk About When I Talk About Platforms. *martinFowler.com*, 2018. Available at: https://martinfowler.com/articles/talk-about-platforms.html. Accessed in: October 11, 2023.

CHRISTENSEN, Henrik. Teaching DevOps and Cloud Computing using a Cognitive Apprenticeship and Story-Telling Approach. In: 2016 ACM Conference on Innovation and Technology in Computer Science Education, Jul. 2016, Arequipa, Peru. *Proceedings...*, p. 174-179, 2016. Available at:

https://dl.acm.org/doi/10.1145/2899415.2899426. Accessed in: 11 Oct. 2023.

CONWAY, Melvin. How do committees invent. In: *Datamation*, v. 14, no. 4, p. 28-31. 1968. Available at: http://www.melconway.com/Home/pdf/committees.pdf. Accessed in: 11 Oct. 2023.

CUKIER, Daniel. DevOps patterns to scale web applications using cloud services. In: Conference on Systems, Programming, & Applications: Software for Humanity (SPLASH'13), Oct. 2013, Indianapolis, Indiana, USA. *Proceedings...*, p. 143-152, 2013. Available at: https://www.researchgate.net/publication/262404654_DevOps_patterns_to_scale_web_applications_using_cloud_services. Accessed in: 11 Oct. 2023.

DAVIS, Jennifer; DANIELS, Ryn. *Effective DevOps:* Building a Culture of Collaboration, Affinity, and Tooling at Scale. Sevastopol (USA): O'Reilly Media, 2016.

DEBOIS, Patrick. *Agile Infrastructure & Operations.* 2008. Slide presentation. Available at: https://docplayer.net/23874035-Agile-infrastructure-

operations.html. Accessed in: Oct. 2022.

FAYOL, Henri. *General and Industrial Management*. Reprint of the 1949 edition. Originally published in 1916. Eastford (USA): Martino Fine Books, 2013.

FEIJTER, Rico de; OVERBEEK, Sietse; VAN VLIET, Rob; JAGROEP, Erik; BRINKKEMPER, Sjaak. DevOps Competencies and Maturity for Software Producing Organizations. In: Enterprise, Business-Process and Information Systems Modeling 2018. *Lecture Notes in Business Information Processing*, v. 318, Springer, Cham, 2018. p. 244-259. Available at: https://link.springer.com/chapter/10.1007/978-3-319-91704-7_16. Accessed in: 11 Oct. 2023.

FORSGREN, Nicole; HUMBLE, Jez; Kim, Gene. Measuring Performance. In: *Accelerate*: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations. Portland (USA): IT Revolution Press, 2018.

FORSGREN, Nicole; KERSTEN, Mik. DevOps metrics. In: *Communications of the ACM*, v. 61, no. 4, p. 44-48. 2018. Available at: https://dl.acm.org/doi/10.1145/3159169. Accessed in: 11 Oct. 2023.

FOWLER, Martin. BlueGreenDeployment. *martinFowler.com*, 2010. Available at: https://martinfowler.com/bliki/BlueGreenDeployment.html. Accessed in: Nov. 2022.

FRIED, Jason; HANSSON, Heinemeier; LINDERMAN, Matthew. Stay Lean. In: *Getting Real*. Chicago (USA): 37signals, 2006. Available at: https://basecamp.com/gettingreal. Accessed in: 11 Oct. 2023.

GALBRAITH, Jay. Organization Design. In: CUMMING, Thomas. (ed.). *Handbook of Organization Development*. Thousand Oaks/CA (USA): Sage Publications Inc., 2008.

GLASER, Barney; STRAUSS, Anselm. *The Discovery of Grounded Theory:* Strategies for Qualitative Research. Originally published in 1967. New Brunswick (USA): Aldine Transaction, 1999.

GRAY, Jim. A conversation with Werner Vogels. *Queue*, v. 4, no. 4, p. 14-22. 2006. Available at: https://queue.acm.org/detail.cfm?id=1142065. Accessed in: 11 Oct. 2023.

GREGOR, Shirley. The Nature of Theory in Information Systems. *MIS Quarterly*, v. 30, no. 3, Sept. 2006, p. 611-642. 2006. Available at: https://misq.umn.edu/skin/frontend/default/misq/pdf/TheoryReview/Gregor.pdf. Accessed in: 11 Oct. 2023.

GOLDRATT, Eliyahu M.; COX, Jeff. *The Goal:* A Process of Ongoing Improvement. 30th Anniversary Edition. Great Barrington/MA (USA): North River Press, 2014.

HUMBLE, Jez; FARLEY, David. *Continuous Delivery:* Reliable Software Releases through Build, Test, and Deployment Automation. Upper Saddle River/NJ (USA): Addison-Wesley Professional, 2010.

HUMBLE, Jez; MOLESKY, Joanne. Why Enterprises Must Adopt Devops to Enable Continuous Delivery. *Cutter IT Journal*, v. 24, no. 8, p. 6-12. 2011. Available at: https://www.uio.no/studier/emner/matnat/ifi/IN5430/v20/pensumliste/readings/humblemolesky_2011_devops.pdf. Accessed in: 11 Oct. 2023.

HUMBLE, Jez. There's No Such Thing as a "Devops Team". *Continuous Delivery*, 2012. Available at:

https://continuousdelivery.com/2012/10/theres-no-such-thing-as-a-devops-team/. Accessed in: Dec. 2022.

HUMBLE, Jez. Continuous Delivery Sounds Great, but Will It Work Here? *Queue*, v. 15, no. 6, Nov-Dec 2017, p. 57-76. 2017. Available at: https://queue.acm.org/detail.cfm?id=3190610. Accessed in: 11 Oct. 2023.

ITIL® FOUNDATION. ITIL 4 Edition Glossary. *Axelos*, 2019. Available at: https://purplegriffon.com/downloads/resources/itil4-foundation-glossary-january-2019.pdf. Accessed in: Oct. 2022.

JESSOP, Bob. Fordism and post-Fordism: a critical reformulation. In: STORPER, Michael; SCOTT, Allen J. (ed.). *Pathways to industrialization and regional development*. Abingdon: Routledge, 1992.

KANG, Hui; LE, Michael, TAO, Shu. Container and Microservice Driven Design for Cloud Infrastructure DevOps. In: 2016 IEEE International Conference on Cloud Engineering (IC2E), Berlin, Germany, 2016, p. 202-211. 2016. Available at: https://doi.ieeecomputersociety.org/10.1109/IC2E.20

16.26. Accessed in: 11 Oct. 2023.

KIM, Gene; HUMBLE, Jez; DEBOIS, Patrick; WILLIS, John. *The DevOps Handbook:* How to Create World-Class Agility, Reliability, and Security in Technology Organizations. Portland/OR (USA): IT Revolution Press, 2016.

KIM, Gene; BEHR, Kevin; SPAFFORD, George. *The Phoenix Project:* A Novel about IT, DevOps, and Helping Your Business Win. 3rd ed. Portland/OR (USA): IT Revolution Press, 2018.

KNIBERG, Henrik. Spotify engineering culture (part 1). *Spotify R&D*, 2014. Available at: https://labs.spotify.com/2014/03/27/spotify-engineering-culture-part-1. Accessed in: Jan. 2023.

KOTTER, John; COHEN, Dan. *The Heart of Change:* Real-Life Stories of How People Change Their Organizations. Brighton/MA (USA): Harvard Business Review Press, 2012.

LEITE, Leonardo; ROCHA, Carla; KON, Fabio; MILOJICIC, Dejan; MEIRELLES, Paulo. A Survey of DevOps Concepts and Challenges. *ACM Computing Surveys*, v. 52, no. 6., p. 1-35. 2019. Available at:

https://arxiv.org/pdf/1909.05409.pdf. Accessed in: 11 Oct. 2023.

LEITE, Leonardo; PINTO, Gustavo; KON, Fabio; MEIRELLES, Paulo. The organization of software teams in the quest for continuous delivery: A grounded theory approach. *Information and Software Technology*, v. 139. 2021. Available at: https://doi.org/10.1016/j.infsof.2021.106672. Accessed in: 11 Oct. 2023.

LEITE, Leonardo. *A grounded theory of organizational structures for development and infrastructure professionals in software-producing organizations*. 2022. Thesis (Doctorate in Computer Science) - Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2022. Available at: https://doi.org/10.11606/T.45.2022.tde-28062022-132626. Accessed in: Oct. 11, 2023.

LEITE, Leonardo; LAGO, Nelson; MELO, Claudia; KON, Fabio; MEIRELLES, Paulo. A Theory of Organizational Structures for Development and Infrastructure Professionals. *IEEE Transactions on Software Engineering*, v. 49, n. 4, p. 1898-1911. 2023. Available at: https://ieeexplore.ieee.org/document/9864071.

Accessed in: 11 Oct. 2023.

LEWIS, James; FOWLER, Martin. Microservices. *martinFowler.com*, 2014. Available at: https://www.martinfowler.com/articles/microservices.html. Accessed in: Nov. 2022.

MAGOUTIS, Kostas; POPOULAS, Christos; PAPAIOANNOU, Antonis; KARNIAVOURA, Flora; AKESTORIDIS, Dimitrios-Georgios; PAROTSIDIS, Nikos; KOROZI, Maria; LEONIDIS, Asterios; NTOA, Stavroula; STEPHANIDIS, Constantine. Design and implementation of a social networking platform for cloud deployment specialists. *Journal of Internet Services and Applications*, v. 6, no. 1. 2015. Available at: https://jisajournal.springeropen.com/articles/10.1186/s13174-015-0033-5. Accessed in: 11 Oct. 2023.

MANNS, Mary; RISING; Beautiful. *Fearless Change:* Patterns for Introducing New Ideas. Boston/MA (USA): Addison-Wesley, 2004.

MARX, Karl. Preface. In: *A Contribution to the Critique of Political Economy*. Originally published in German in 1859. Delhi: Lector House, 2020.

MELL, Peter; GRANCE, Timothy. The NIST definition of cloud computing. *NIST*, 2011. Available at: http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf. Accessed in: Sep. 2023.

MORGAN, Gareth. Reflections on Images of Organization and Its Implications for Organization and Environment. *Organization & Environment*, v. 24, no. 4, p. 459-478. 2011. Available at: https://doi.org/10.1177/1086026611434274. Accessed in: 11 Oct. 2023.

NYBOM, Kristian; SMEDS, Jens; PORRES, Ivan. On the Impact of Mixing Responsibilities Between Devs and Ops. In: International Conference on Agile Software Development (XP 2016), 2016, Edinburgh, UK. *Lecture Notes in Business Information Processing*, vol. 251, p. 131-143. Springer International Publishing, 2016.

OHNO, Taiichi. *Toyota Production System:* Beyond Large-Scale Production. Originally published in Japanese in 1978. New York (USA): Productivity Press, 1988.

OLIVEIRA, Nelio. Organizational Structure, Format, Shape, Design and Architecture. In: *Automated*

Organizations: Development and Structure of the Modern Business Firm. Berlin: Physica-Verlag, 2012.

PENZENSTADLER, Birgit; DUBOC, Leticia; VENTERS, Colin; BETZ, Stefanie; SEYFF, Norbert; WNUK, Krzysztof; CHITCHYAN, Ruzanna; EASTERBROOK, Steve; BECKER, Christoph. Software Engineering for Sustainability: Find the Leverage Points! *IEEE Software*, v. 35, no. 4, Jul-Aug 2018, p. 22-33. 2018. Available at: https://www.cs.toronto.edu/~sme/papers/2018/Penzenstadler-etal-2018.pdf. Accessed in: 11 Oct. 2023.

PFLAEGING, Niels. *Organize for Complexity:* How to Get Life Back Into Work to Build the High-Performance Organization. 5. ed. (s. l.): Betacodex Publishing, 2014.

POPPENDIECK, Mary; POPPENDIECK, Tom. *Implementing Lean Software Development:* From Concept to Cash. Upper Saddle River/NJ (USA): Addison-Wesley Professional, 2006.

RALPH, Paul. Toward Methodological Guidelines for Process Theories and Taxonomies in Software Engineering. *IEEE Transactions on Software Engineering*, v. 45, no. 7, p. 712-735. 2019. Available

at: https://ieeexplore.ieee.org/document/8267085. Accessed in: 11 Oct. 2023.

ROBERTS, Mike. Serverless Architectures. *martinFowler.com*, 2018. Available at: https://martinfowler.com/articles/serverless.html. Accessed in: Oct. 2022.

ROTHER, Mike. *Toyota Kata:* Managing People for Improvement, Adaptiveness and Superior Results. New York (USA): McGraw-Hill, 2009.

SATO, Danilo. CanaryRelease. *martinFowler.com*, 2014. Available at: https://martinfowler.com/bliki/CanaryRelease.html. Accessed in: Nov. 2022.

SENGE, Peter; KLEINER, Art; ROBERTS, Charlotte; ROSS, Richard; ROTH, George; SMITH, Bryan. *The Dance of Change:* The Challenges to Sustaining Momentum in a Learning Organization. New York (USA): Broadway Business, 1999.

SIQUEIRA, Rodrigo; CAMARINHA, Diego; WEN, Melissa; MEIRELLES, Paulo; KON, Fabio. Continuous Delivery: Building Trust in a Large-scale, Complex Government Organization. *IEEE Software*, v. 35, no.

2, p. 38-43. 2018. Available at: https://www.ime.usp.br/~kon/papers/IEEESoftware2018.pdf. Accessed in: 11 Oct. 2023.

SKELTON, Matthew; PAIS, Manuel. *Team Topologies:* Organizing Business and Technology Teams for Fast Flow. Portland/OR (USA): IT Revolution, 2019.

SINCLAIR, Toby. 12 Organizational Design Principles that Embrace Complexity. *Toby Sinclair*, 2020. Available at: https://www.tobysinclair.com/post/organisational-design-principles-that-embrace-complexity. Accessed in: May 2023.

SMITH, Adam. Of the division of labour. In: *The Wealth of Nations*. Originally published in 1776. New York: Barnes & Noble Classics, 2004.

SNYDER, Barry; CURTIS, Bill. Using Analytics to Guide Improvement during an Agile–DevOps Transformation. *IEEE Software*, v. 35, n. 1, p. 78-83. 2018. Available at: https://doi.org/10.1109/MS.2017.4541032. Accessed on 11 Oct. 2023.

SOSA, Manuel; EPPINGER, Steven; ROWLES, Craig.

The Misalignment of Product Architecture and Organizational Structure in Complex Product Development. *Management Science*, v. 50 n. 12, p. 1674-1689. 2004. Available at: https://doi.org/10.1287/mnsc.1040.0289. Accessed in: 11 Oct. 2023.

VASILESCU, Bogdan; BLINCOE, Kelly; XUAN, Qi; CASALNUOVO, Casey; DAMIAN, Daniela; DEVANBU, Premkumar; FILKOV, Vladimir. The Sky Is Not the Limit: Multitasking Across GitHub Projects. In: 38th INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), 2016, Austin, Texas, USA. *Proceedings...*, p. 994-1005. 2016. Available at: https://doi.org/10.1145/2884781.2884875. Accessed in: 11 Oct. 2023.

WOODS, Eoin. Operational: The Forgotten Architectural View. *IEEE Software*, v. 33, no. 3, p. 20-23. 2016. Available at: https://ieeexplore.ieee.org/document/7458767. Accessed in: 11 Oct. 2023.

YEATTS, Dale; HYTEN, Cloyd. *High-Performing Self-Managed Work Teams:* A Comparison of Theory to Practice. Thousand Oaks/CA (USA): Sage Publications, 1998.